THE PETER NORTON PROGRAMMING LIBRARY

BASIC AVANZATO

- Più di 100 programmi pronti all'uso
- Consigli utili per la creazione di progetti grafici professionali, utilizzo di linguaggi diversi, programmi che usano il mouse, e molto altro
- QuickBASIC e il BASIC Professional Development System

Steven Holzner e
Peter Norton Computing, Inc.

GRUPPO EDITORIALE JACKSON
Brady

BASIC AVANZATO

Steven Holzner e
Peter Norton Computing, Inc.

GRUPPO EDITORIALE
JACKSON
Viale Restelli 5 - 20124 Milano

Brady

Titolo originale
ADVANCED BASIC

Original English Language edition published by BRADY PUBLISHING
A Division of Simon & Schuster, Englewood Cliffs, New Jersey, USA

Copyright per l'edizione originale in lingua inglese
© 1991, Peter Norton

Una coedizione di
Prentice Hall International, Hemel Hempstead, England
Gruppo Editoriale Jackson, Milano, Italia

Copyright per la traduzione in lingua italiana
© 1992, Prentice Hall International - Gruppo Editoriale Jackson

Traduzione
Claudio Nasso

Impaginazione con tecniche di desktop publishing
L'Angolo Grafico - 20090 San Felice Segrate (MI)

Redattore di collana
Daniele Cassanelli

Copertina
Laura Berettini

Tutti i diritti sono riservati. Nessuna parte di questo libro può essere riprodotta, memorizzata in sistemi d'archivio, o trasmessa in qualsiasi forma o mezzo, elettronico, meccanico, fotocopia, registrazione o altri, senza la preventiva autorizzazione scritta dell'editore.

Gli autori e l'editore di questo volume si sono fatti carico della preparazione del libro e dei programmi in esso contenuti. Questa attività ha compreso la ricerca, lo sviluppo e il test di teorie e di programmi per determinare le loro funzionalità. Gli autori e l'editore non si assumono alcuna responsabilità, esplicita o implicita, riguardante questi programmi o il contenuto del testo. Gli autori e l'editore non potranno in alcun caso essere ritenuti responsabili per incidenti o conseguenti danni che derivino o siano causati dall'uso dei programmi o dal loro funzionamento.

Prima edizione 1992

Ristampa

9 8 7 6 5 4 3 2 1

1996 1995 1994 1993 1992

INDICE

A chi è destinato questo libro IX
Contenuto del volume IX
Due parole sulle peter norton microcomputer libraries IX
Nota dell'editore X
Marchi registrati X

INTRODUZIONE XI
Un gradito ritorno del BASIC XI
Il Basic Professional Development System XII
Il BASIC è un linguaggio di programmazione molto serio XIII
Contenuto del libro XIII
Materiale necessario XV
Nota all'edizione italiana XVI

Capitolo 1 - CORRETTA IMMISSIONE DEI PROGRAMMI 1
L'istruzione INPUT 1
La funzione INPUT$ 2
L'istruzione LINE INPUT 4
INKEY$ 4
GetArrowKey$ – Lettura dei tasti ↑ ↓ ← → 5
InKeyNoEcho$ 9
Uso degli interrupt BIOS e DOS 9
Un servizio DOS per la tastiera 15
Scrittura di una funzione di immissione 19
Interpretazione delle immissioni da tastiera 29
Conclusione 36

Capitolo 2 - FINESTRE 37
Progettazione di un sistema personalizzato di finestre 37
Inizializzazione del sistema personalizzato a finestre 38
Compilazione dell'array delle finestre 41
Esempio di inizializzazione di una finestra 46
Visualizzazione di una finestra 49
Prova di visualizzazione di una finestra 56
Rimozione di una finestra 57
Prova di rimozione di una finestra 60
Stampa di un testo in una finestra 61
Prova di stampa di un testo in una finestra 71
Spostamento di finestre 72
Prova di spostamento di finestre 80
Cancellazione di una finestra 80
Prova di cancellazione di una finestra 84
Il BASIC PDS e le finestre 84
WINDOWER.BAS: un programma a finestre di BASIC PDS 85
Conclusione 89

Capitolo 3 - MOUSE 91
Prima di iniziare 91
Inizializzazione del mouse 92
La funzione MouseInitialize%( ) 93
Esempio di inizializzazione del mouse 94
Visualizzazione del puntatore del mouse 94
Il sottoprogramma MouseShowCursor( ) 94
Uso di MouseShowCursor( ) 95
Come nascondere il puntatore del mouse 96
Il sottoprogramma MouseHideCursor( ) 97
Uso del sottoprogramma MouseHideCursor( ) 98
Lettura immediata delle informazioni del mouse 99
Il sottoprogramma MouseInformation( ) 99
Controllo dello stato del mouse 102
Spostamento del puntatore del mouse 104
Il sottoprogramma MouseMoveCursor( ) 104
Spostamento del puntatore all'angolo superiore sinistro dello schermo 105
Lettura della "coda" dei pulsanti premuti 106
Il sottoprogramma MouseTimesPressed( ) 107
Conteggio delle pressioni di un tasto 109
Lettura della coda dei rilasci di un pulsante del mouse 110
Il sottoprogramma MouseTimesReleased( ) 110
Conteggio dei rilasci del pulsante 112
Limitazione degli spostamenti orizzontali 113
Il sottoprogramma MouseHorizontalRange( ) 114
Utilizzo di MouseHorizontalRange( ) 115
Limitazione degli spostamenti verticali 116

Il sottoprogramma MouseVerticalRange( ) 116
Utilizzo di MouseVerticalRange( ) 117
Impostazione del puntatore del mouse 118
La maschera video e la maschera puntatore 118
Trasformazione del puntatore in un punto 123
Il mouse con il BASIC PDS 124
Conclusione 126

Capitolo 4 - MENU A TENDINA 127
Progettazione di un sistema di menu a tendina 127
Considerazioni particolari sul sistema a menu 129
Inizializzazione del sistema a menu 130
La funzione MenuInitialize%( ) 133
Prova di inizializzazione di una barra dei menu 140
VIsualizzazione della barra dei menu 141
Prova di visualizzazione della barra dei menu 147
Rimozione della barra dei menu 149
Prova di rimozione della barra dei menu 152
Lettura delle immissioni da menu 154
Prova di attesa di immissioni dai menu 175
Lettura di immissione da menu senza attesa 177
Prova di immissioni provenienti dal menu 186
Marcatura di una selezione da menu 188
Prova di marcatura di voci nel menu 190
Disattivazione della marcatura di una voce di menu 192
Prova di disattivazione di voci del menu 193
Controllo di un menu 196
Prova di lettura delle voci del menu 198
Il sistema a menu del BASIC PDS 199
Conclusione 210

Capitolo 5 - GRAFICA 211
Un programma paint gestito dal mouse 212
Subroutine INITIALIZE 216
La subroutine GetLeftButtonPress 218
La subroutine MenuChoice 219
La subroutine DrawPixel 228
La subroutine DrawLine 230
La subroutine DrawBox 233
La subroutine DrawCircle 234
La subroutine DrawPaint 236
Il listato completo del programma PAINT 236
Disegno di un orologio 242
Sprite e animazione 244
La subroutine DrawSprite 250
La subroutine GetSprite 251
La subroutine SaveSprite 253

La subroutine LoadSprite 254
Il listato completo del programma SPRITE.BAS 254
Presentation Graphics del BASIC PDS 256
Grafici a torta 257
I grafici a barre 263
I grafici a linee 264
Controllo della scheda grafica installata 266
Controllo della risoluzione verticale e orizzontale del video 271
Conclusione 273

Capitolo 6 - DATABASE 275
L'indice corrente del database 278
Un programma di database 281
Alla ricerca dei record 295
Cancellazione di record 301
Aggiornamento di file ISAM 304

Capitolo 7 - TECNICHE AVANZATE DI GESTIONE E ORDINAMENTO DEI DATI 313
Le variabili del BASIC 313
L'istruzione DATA 314
Matrici 316
Strutture di dati 319
Liste concatenate 320
Buffer circolari 326
Alberi binari 327
SHELL SORT 336
QUICKSORT 347
Alla ricerca dei dati 354
Conclusioni 361

Capitolo 8 - DEBUGGING 363
DEBUGGING IN QUICKBASIC 363
CODEVIEW 374

Capitolo 9 - ANALISI DEL BIOS E DEL DOS 387
Gli interrupt del BIOS 388
Il modo schermo del BIOS 389
Come determinare l'hardware installato nel computer 392
Gli interrupt del DOS 396
Impostazione del disco di default 397
Come ottenere il disco di default 400
Determinare lo spazio libero 402
I servizi del DOS per i file 405
Trovare il primo file che corrisponde ai requisiti 406
Per trovare i file successivi 415

Il resto del DOS 419
Conclusione 420

Capitolo 10 - IL LINGUAGGIO ASSEMBLY 421
Linguaggio macchina 421
Linguaggio Assembly 423
L'istruzione MOV 423
Un esempio in linguaggio Assembly 424
Il primo programma in linguaggio Assembly 428
Segmentazione della memoria 431
Segmenti in memoria 432
I registri di segmento al lavoro:
I file .COM 435
Direttive per l'Assembler 436
La direttiva .CODE 436
Etichette 437
Posizionamento del codice nel segmento del codice 437
La direttiva END 438
Assemblaggio di STAMPAZ.ASM 439
Aggiungere dati ai programmi in Assembly 440
Le direttive DB e DW 440
Stringhe in memoria 443
Il servizio 9 dell'INT 21H:
Stampa una stringa 444
L'uso dei commenti 446
Accettare input da tastiera 446
Il programma CAP.COM 446
Le istruzioni SUB e ADD 448
Salti condizionati 450
L'istruzione CMP 450
Ancora sui salti condizionati 452
Un programma per convertire da esadecimale a decimale 453
Le istruzioni DIV e MUL 461
L'istruzione LOOP 468
Procedure in linguaggio Assembly 469
Conclusione 474

Capitolo 11 - INTERFACCIAMENTO TRA BASIC E LINGUAGGIO ASSEMBLY 475
Collegamento al BASIC 476
Passaggio dei parametri dal BASIC 478
Restituzione di valori al BASIC 480
La rappresentazione interna dei dati del BASIC 481
INTEGER e LONG 482
Notazione in complemento a due 482
SINGLE e DOUBLE 484

CURRENCY 486
Stringhe e matrici 486
Una funzione senza parametri 489
Una funzione con un parametro 491
Una funzione con un parametro INTERO LONG 493
Una funzione che restituisce una stringa 494
GETQQQ$( ) con QB.EXE e BC.EXE 495
GETQQQ$( ) con QBX.EXE 498
Una funzione con due parametri 502
Una funzione che usa il segmento di dati 503
Un sottoprogramma con parametri 505
Un sottoprogramma con un parametro stringa 507
PRINTSTRING( ) con BC.EXE e QB.EXE 507
PRINTSTRING( ) con QBX.EXE 509
Un sottoprogramma con un parametro matrice 512
PRINTFIRSTELEMENT con BC.EXE 513
PRINTFIRSTELEMENT con QB.EXE e QBX.EXE 514
Conclusioni 515

Appendice - REFERENCE BIOS E DOS 517
Interrupt del BIOS 517
Interrupt del DOS 534

A CHI È DESTINATO QUESTO LIBRO

A tutti i programmatori, principianti o esperti, che utilizzano il BASIC e che intendono approfondire le proprie conoscenze di programmazione o che desiderano incrementare le funzioni dei propri programmi.

CONTENUTO DEL VOLUME

- Più di 100 programmi pronti per essere avviati che mostrano come operare con le finestre, con i menu a tendina, con l'animazione, con il mouse e che presentano nuove e veloci procedure.
- Consigli di tipo avanzato che saranno in grado di velocizzare e integrare i programmi già scritti in BASIC.
- Una tecnica pratica di approccio alla programmazione che mostra l'azione dei vari codici utilizzando uno stile diretto e di facile comprensione.

DUE PAROLE SULLE PETER NORTON MICROCOMPUTER LIBRARIES

Tutti i volumi della collana *Peter Norton Microcomputer Libraries* sono stati realizzati in collaborazione con la *Peter Norton Computing*, e contengono approfondite argomentazione sui più recenti sviluppi dell'hardware, dei sistemi operativi e della programmazione. I volumi sono stati controllati e rigorosamente verificati dagli esperti della *Peter Norton Computing* e per questa ragione questi sono libri che dovrebbero sempre essere presenti nella libreria degli operatori che fanno uso di microcomputer. La collana è composta da due serie di volumi:

- *The Peter Norton Hardware Library* che prende in esame le componenti e l'operatività dei microcomputer.
- *The Peter Norton Programming Library* che esamina in dettaglio la creazione e la corretta gestione di programmi con vari tipi di linguaggi di programmazione.

NOTA DELL'EDITORE

Gli autori e l'editore originale di questo volume hanno fatto del loro meglio per verificare la corretta realizzazione di questo libro e dei programmi in esso contenuti concentrando la propria attenzione sullo sviluppo, la ricerca e la verifica di tutti i concetti e del contenuto dei programmi per poterne garantire la loro effettiva correttezza. Gli autori e l'editore non forniscono tuttavia alcun tipo di garanzia, esplicita o implicita, sui programmi e la documentazione contenuta in questo volume. L'autore e l'editore non verranno quindi mai ritenuti responsabili per eventuali danni derivanti dall'uso occasionale o consequenziale che dovessero presentarsi in relazione all'uso o alla funzionalità dei programmi contenuti in questo libro.

MARCHI REGISTRATI

Basic e QuickBASIC sono marchi registrati della Microsoft Corporation.
OS/2 è un marchio registrato della IBM Corporation.
Logitech mouse è un marchio registrato della Logitech. Inc.

INTRODUZIONE

UN GRADITO RITORNO DEL BASIC

Per molti anni, il BASIC è stato considerato un linguaggio non destinato ai programmatori, ma ai soli dilettanti. Il BASIC (oppure il BASICA o il GWBASIC), in effetti era un linguaggio che veniva sempre fornito gratuitamente con il computer ed è sempre stato considerato un linguaggio di apprendimento in grado di far apparire a video, per esempio, la parola "Salve!". In effetti, se ci si addentrava in un tipo di programmazione complesso e più approfondito, si scopriva ben presto che i tempi di elaborazione erano relativamente lenti e molto spesso venivano generati messaggi di errore del tipo "Redo from Start". Il BASIC, quindi, è stato sempre considerato dai programmatori la *Cenerentola* dei linguaggi di programmazione.

Tuttavia le situazioni possono sempre cambiare soprattutto quando si parla di velocità di elaborazione. Il BASIC non è stato immune da questo processo di cambiamento e offre ora un'inaspettata vitalità collegata a compilatori molto veloci. I codici eseguibili che venivano generati da questi compilatori erano, se non i più veloci, a un buon livello di velocità; a mano a mano che venivano rilasciate nuove versioni la velocità di esecuzione del codice è sempre aumentata e i programmatori della Microsoft hanno ripreso a creare importanti frammenti di codice utilizzando il BASIC. Indubbiamente, queste e altre innovazioni hanno cominciato a stimolare i programmatori in ambiente PC consacrando ufficialmente il ritorno del BASIC.

Con la riduzione dei tempi di esecuzione, i programmatori hanno iniziato nuovamente ad apprezzare alcune caratteristiche salienti del BASIC: il suo esteso numero di istruzioni utilizzabili e la sua incontestata praticità che non trova pari in nessun altro linguaggio di programmazione per PC. Anche la più recente versione di Microsoft BASIC è in grado di gestire ben 160 tra funzioni, enunciati e comandi. In

sostanza, il BASIC è in grado di racchiudere in una sola riga le istruzioni che, con altri linguaggi di programmazione, richiederebbero molte più righe di istruzione.
Durante il proprio sviluppo, il BASIC ha assorbito molti elementi da altri linguaggi; per esempio, dal *Pascal* ha preso in prestito la struttura SELECT CASE, dal *FORTRAN* ha preso in prestito l'istruzione COMMON e dal *C* ha assorbito un uso rudimentale dei puntatori (VARPTR, VARSEG, SSEG, SADD, SSEGADD, ecc.).
L'ormai nota istruzione *IF* è stata arricchita da una struttura del tipo *IF THEN ... ELSE ... END IF* e ora l'utente è in grado di definire i tipi di dati. È stato introdotto il sistema per accedere direttamente al sistema operativo (con l'uso di *INTERRUPT( )* e *INTERRUPTX( )*): in conclusione, un linguaggio che sembrava possedere ormai tutto, è stato ancora migliorato.

IL BASIC PROFESSIONAL DEVELOPMENT SYSTEM

Nel novembre del 1989, il rilascio di BASIC 7.0 (conosciuto anche con il nome di *BASIC Professional Development System* (PDS)) ha consentito il raggiungimento di un'altra pietra miliare. Lo scopo della versione 7.0 era quello di divenire un serio strumento di programmazione (a un prezzo di vendita anch'esso molto serio) e ha confermato l'orientamento della Microsoft verso il BASIC in generale. Nella versione 7.0 sono state aggiunte librerie che contengono funzioni di tipo finanziario e funzioni di formattazione avanzata; vi è stato anche incluso un intero elenco di strumenti per la generazione di matrici matematiche, per la presentazione di grafici e un'interfaccia utente di tipo avanzato; sono questi elementi che verranno presi in esame nel corso di questo libro.
Inoltre, è stata inserito un nuovo tipo di struttura dei file utilizzabile per i database, i file ISAM. Per quanto riguarda i puntatori, i record ISAM potranno essere rapidamente e facilmente ordinati garantendo una gestione dei database di tipo professionale. Il BASIC continua incessantemente a crescere; in effetti, *prima* di presentare queste novità, era ed è in grado di gestire non meno di 252 funzioni, istruzioni, comandi, metacomandi e routine "preconfezionate" e di pari passo con la crescita del BASIC cresce anche il numero di programmatori che utilizzano questo linguaggio.
In breve, il BASIC, con il suo ritorno, si posiziona nuovamente ai primi posti della classifica dei linguaggi di programmazione.

IL BASIC È UN LINGUAGGIO DI PROGRAMMAZIONE MOLTO SERIO

Ora che il BASIC viene regolarmente utilizzato da seri professionisti e programmatori, questa serietà viene accompagnata da una regolare produzione di documentazione editoriale. Non verranno più prodotti libri con capitoli che si dilungano in interminabili dissertazioni sul set di istruzioni del BASIC o che contengono elenchi alfabetici dei comandi e delle istruzioni BASIC che assomigliano più a dizionari che a trattati completi ed esaustivi. La tendenza odierna, invece, mira a fornire al programmatore e al professionista una serie di informazioni e procedure che sviluppino e arricchiscano le nozioni già in possesso del lettore: in effetti è questo che un serio professionista richiede a un trattato sul BASIC.

Questo concetto è ancora più vero se si intende operare sul BASIC di tipo avanzato poiché un libro di questo tipo è e deve essere destinato a un pubblico di programmatori il cui scopo finale è quello di sfruttare al meglio il tempo a loro disposizione, senza alcuna dispersione di energie. Per questa ragione il contenuto dei programmi che verranno presentato prende in esame i più importanti e attuali concetti di programmazione. Verranno presentati tutti i trucchi che un programmatore dovrebbe conoscere, come, per esempio, l'applicazione ai programmi della gestione del mouse e dei menu a tendina, o l'interfacciamento di routine in linguaggio assembly direttamente con il BASIC. Per rendere più agevole il lavoro di programmazione, in questo libro verranno utilizzate anche alcuni sottoprogrammi interni del BASIC.

Man mano che verrà analizzato il BASIC di tipo avanzato, verranno anche presi in considerazione tutti quegli strumenti che vengono messi a disposizione dal BASIC Professional Development System. Un intero capitolo, il sesto, viene dedicato esclusivamente alla programmazione PDS: qui verrà creato un programma di database che utilizza il nuovo sistema di file ISAM che dimostrerà la velocità e la semplicità d'utilizzo del BASIC Professional Development System.

Per concludere, in questo libro vengono riportate tutte le più recenti innovazioni apportate al BASIC.

CONTENUTO DEL LIBRO

Si è deliberatamente puntata l'attenzione sugli *esempi* e sul loro progressivo sviluppo per dare la possibilità al lettore di comprendere appieno ciò su cui sta operando. Per indicare a che punto del listato ci si trova verrà utilizzato il sottolineato e, inoltre, verranno presentati una serie di consigli che faciliteranno la corretta comprensione della programmazione in BASIC. Quando si presenterà l'occasione di trattare alcuni concetti *storici* o strettamente legati all'*hardware* che è indispensabile conoscere, questi verranno messi in evidenza.

Di seguito viene fornito un breve riassunto del contenuto di ciascun capitolo.

Introduzione Panoramica sul contenuto del libro e materiale necessario per utilizzare i programmi presentati.

Capitolo 1 *Corretta immissione dei programmi.* Viene analizzata la corretta immissione dei programmi in modo che questi abbiano sempre un aspetto di tipo professionale. Si vedrà come gestire l'immissione in modo professionale e come inserire gli elementi che il BASIC non ha preso in considerazione (come, per esempio, una versione personalizzata di INKEY$ che non visualizzi sullo schermo i caratteri che viene richiesto di digitare).

Capitolo 2 *Finestre.* Viene preso in esame lo sviluppo del sistema a finestre. Tra i molti esempi riportati molti esempi e si vedrà come impostare una finestra, come visualizzarla sullo schermo, come stamparla, come spostarla, ridimensionarla o rimuoverla. Il BASIC Professional Development System (PDS) contiene alcune funzioni predisposte per la gestione delle finestre.

Capitolo 3 *Mouse.* Verrà presa in esame la gestione del mouse attraverso l'interrupt &H33. Si vedrà come visualizzare o nascondere il puntatore del mouse e come attivare la lettura di informazioni con poche e semplici operazioni. Per coloro che utilizzano il PDS si prenderà in esame l'interfaccia predisposta per la gestione del mouse.

Capitolo 4 *Menu a tendina.* In questo capitolo viene presa in esame la contemporanea operatività di mouse e finestre per la creazione di menu a tendina. Si vedrà come generare una riga dei menu dalla quale aprire menu a tendina servendosi sia della tastiera che del mouse. Anche in questo caso, per gli utenti che fanno uso del PDS verranno analizzati alcuni strumenti collegabili ai menu.

Capitolo 5 *Grafica.* Viene analizzato come aumentare le possibilità grafiche del BASIC per poter generare un programma grafico personalizzato. In questo capitolo, si vedrà anche come generare e utilizzare un ambiente grafico e si analizzeranno le presentazioni grafiche del PDS che comprendono, tra l'altro, le animazioni.

Capitolo 6 *Database.* In questo capitolo verrà realizzato un programma di database e si vedrà come i file ISAM del PDS siano in grado di facilitare la programmazione di un database.

Capitolo 7 *Tecniche avanzate di gestione e ordinamento dei dati.* Questo capitolo analizza l'ordinamento dei dati di un array. Per avviare questa operazione sarà sufficiente utilizzare una delle varie rou-

tine di ordinamento. Inoltre, si vedrà come utilizzare tutte le tecniche di ordinamento dei dati in BASIC e come sviluppare un programma che esegua una rapida ricerca dei dati.

Capitolo 8 — *Debugging.* Questo capitolo fornisce un'introduzione ai concetti fondamentali di debug, sia in ambiente QuickBasic che tramite l'uso del programma di debug CodeView.

Capitolo 9 — *Analisi del BIOS e del DOS.* Vengono analizzati alcuni concetti collegati al DOS e al BIOS. Tra l'altro, si vedrà come determinare la configurazione hardware, quanto spazio disponibile vi è su disco e come effettuare all'interno del disco fisso una ricerca di file che risponda a determinati criteri (compreso l'uso dei caratteri jolly).

Capitolo 10 — *Il linguaggio assembly.* In questo capitolo vengono presi in considerazione alcuni concetti del linguaggio assembly. I programmatori esperti avranno spesso necessità di ottenere codici realmente rapidi e compatti e non vi è nulla che possa competere con il linguaggio assembly. Infine vengono analizzati i concetti principali del linguaggio assembly in modo da poterli interfacciare al BASIC come viene spiegato nel successivo capitolo.

Capitolo 11 — *Interfacciamento tra BASIC e linguaggio assembly.* Questo capitolo mostra come collegare le routine del linguaggio assembly al BASIC in modo che queste siano esattamente simili a FUNCTION e SUB; inoltre verrà illustrato l'uso delle nuove e semplificate direttive di segmento di MASM 5.1 e versioni successive.

Appendice — *Reference BIOS e DOS.* L'appendice consente di collegare le funzioni DOS e BIOS. Nel corso del volume si è fatto uso dei servizi DOS e BIOS e questa appendice ne contiene un completo riepilogo.

MATERIALE NECESSARIO

Ora che si è pronti per iniziare sarà prima necessario conoscere il materiale richiesto per poter utilizzare i programmi presentati in questo libro. Sarà necessario dotarsi di un prodotto Microsoft BASIC come, per esempio il compilatore Microsoft BASIC (BC.EXE), il QuickBASIC (QB.EXE oppure il QuickBASIC esteso, QBX.EXE), oppure il BASIC Professional Development System (PDS) (BC.EXE Versione 7.0, 7.10 o successive).

Inoltre, se si vuole utilizzare il materiale presentato nel Capitoli 10 e 11 per rendere più potenti i programmi BASIC associandoli al linguaggio assembly, sarà necessario anche dotarsi di Microsoft Macro Assembler (MASM.EXE), versione 5.1 o successiva. Viene richiesta la presenza di questa o di successive versioni poiché sono le sole in grado di gestire le direttive semplificate di segmento che hanno apportato importanti modifiche per la programmazione con linguaggi misti.
È anche necessario sapere che molti dei programmi presentati in questo libro accederanno direttamente al BIOS e al DOS tramite le routine INTERRUPT( ) e INTERRUPTX( ) che renderanno disponibile un'ampia risorsa di elaborazione, tenendo presente che qualsiasi cosa faccia il computer viene gestito da queste routine. Se si intende utilizzare QuickBASIC (QB.EXE o QBX.EXE), tuttavia, sarà necessario utilizzare l'opzione /L (come nell'esempio che segue) per avviare programmi che fanno uso di INTERRUPT( ) o INTERRUPTX( ).

```
QB PROG /L
```

oppure

```
QBX PROG /L
```

L'opzione /L caricherà in memoria anche le librerie Quick QB.QLB o QBX.QLB. Quando si lavorerà con il linguaggio assembly nell'ultima parte di questo libro, si vedrà anche come generare librerie Quick di tipo personalizzato che risiederanno al di fuori delle procedure del linguaggio assembly.
Inoltre, i programmi che faranno uso dei INTERRUPT( ) o INTERRUPTX( ) — e i moduli in linguaggio assembly che verranno scritti — non sono in grado di girare in modalità protetta OS/2 (anche se questo non è vero per altri programmi presentati in questo libro) poiché utilizzano una struttura di interrupt del BIOS e del DOS. Tuttavia questi programmi saranno in grado di girare sotto OS/2 in modalità compatibile DOS.

NOTA ALL'EDIZIONE ITALIANA

Per la realizzazione dell'edizione italiana, sono state adottate determinate convenzioni che vengono di seguito riportate per facilitare la comprensione di alcuni elementi all'interno del libro.

- Il testo da digitare viene riportato in carattere diverso dal testo: **Courier neretto.** Per esempio:

```
PRINT "Il programma calcola la media di tre numeri."
INPUT "I numeri"; A!, B!, C!
PRINT "Grazie. La media è:" (A! + B! + C!)/3.0
```

- L'emissione a video del risultato di un programma viene anch'esso riportato in carattere diverso dal testo: `Courier chiaro`. Se all'interno di un'emissione a video viene richiesto di digitare uno o più dati, questi verranno riportati in **`Courier neretto`**. Per esempio:

  ```
  R: > INPUT
  Questo programma calcola la media di tre numeri.
  I numeri? 3 3 3
  ```

  L'emissione a video sopra riportata prevede l'immissione, da parte dell'utente di *3 3 3*.

- Quando nel contesto di un paragrafo viene indicata l'immissione di dati, anche questa verrà indicata con un **`Courier neretto`**.

- All'interno dei listati viene riportato in sottolineato il punto a cui si fa riferimento nel paragrafo esplicativo di testo che lo precede o lo segue. Per esempio:

  ```
  PRINT "Digitare una stringa di 10 caratteri."
  Instring$ = INPUT$(10)   ' Nota: nessuna emissione a video.
  VALUE% = INSTR(Instring$, "e")
  IF VALUE% = 0 THEN
   PRINT "Grazie. Nella stringa non vi è alcuna 'e'."
  ELSE
   PRINT "Grazie. La prima lettera 'e' corrisponde"
   PRINT " al carattere" VALU$
  END IF
  ```

- Quando, per ragioni strettamente tipografiche, una riga di listato viene inviata a capo, viene immesso il simbolo ⌉. La presenza di questo simbolo indica che *non* si deve premere **INVIO**, ma che è necessario continuare la digitazione sulla riga corrente.

CAPITOLO 1

CORRETTA IMMISSIONE DEI PROGRAMMI

Questo capitolo prende in esame le procedure BASIC per una corretta immissione dei dati da tastiera. Vengono analizzate varie funzioni e ne vengono create alcune personalizzate per poter interfacciare direttamente con il DOS.
Inoltre si vedrà quali sono le routine necessarie per poter sviluppare una reale applicazione che sia in grado di leggere valori interi che vengono immessi direttamente da tastiera.
Questo capitolo è la naturale introduzione a questo libro poiché insegna come generare immissioni di dati all'interno dei programmi che vengono scritti in BASIC: si inizierà con una breve panoramica di quanto il BASIC stesso è in grado di fornire.

L'ISTRUZIONE *INPUT*

Il listato 1.1 presenta il primo programma di questo libro.

```
PRINT "Il programma calcola la media di tre numeri."
INPUT "I numeri"; A!, B!, C!
PRINT "Grazie. La media è:" (A! + B! + C!)/3.0
```

Listato 1.1. *Programma di calcolo della media.*

Nell'esempio sopra riportato viene utilizzata l'istruzione INPUT che legge i tre valori che vengono digitati e li assegna a tre variabili a precisione semplice (*A!*, *B!* e *C!*).

Nella stessa riga di istruzione è stato specificato il messaggio che si vuole visualizzare a video seguito dall'elenco delle variabili (si veda il Listato 1.1).
L'istruzione *INPUT*, purtroppo, dovrebbe essere evitata dai programmatori esperti poiché questa immette obbligatoriamente un punto interrogativo alla fine della stringa contenente il messaggio. Provare ora a modificare il programma precedente in modo che questo appaia come riportato nel Listato 1.2.

```
INPUT "Immettere i tre numeri"; A!, B!, C!
PRINT "Grazie. La media è:"; (A! + B! + C!)/3
```

Listato 1.2 *Variazione del programma di calcolo della media.*

L'avvio di questo programma genera un'emissione a video simile a quella di seguito riportata:

```
R:\    INPUT
Immettere i tre numeri?
```

Indubbiamente, quanto appare a video non è poi così piacevole. Il vero problema è dato dal fatto che l'istruzione *INPUT* produce un messaggio di errore *Redo from start* nel caso in cui i valori immessi non possano essere indirizzati alla variabile dichiarata oppure se i valori immessi non sono stati separati da delimitatori accettabili (in questo caso la virgola). Per esempio, se nel programma non fossero stati immessi i corretti delimitatori, si sarebbe ottenuto un risultato simile a quello di seguito riportato.

```
R: > INPUT
Questo programma calcola la media di tre numeri.
I numeri? 3 3 3

Redo from start
I numeri?
```

Indubbiamente quanto sopra riportato non è certo quello che si potrebbe definire una piacevole emissione a video e, soprattutto, non rappresenta un'immagine molto professionale del risultato di un programma.

LA FUNZIONE *INPUT$*

Anche se la funzione *INPUT$( )* viene di solito utilizzata per leggere stringhe da un file, essa potrà essere utilizzata anche per leggere i tasti che vengono premuti sulla tastiera. Purtroppo, in questo caso, sarà necessario specificare *esattamente* come dovranno essere letti i caratteri immessi. Di seguito viene riportato un esempio nel

quale verranno letti esattamente 10 caratteri e verrà reperita la prima lettera *e* che compare nel testo digitato.

```
PRINT "Digitare una stringa di 10 caratteri."
Instring$ = INPUT$(10)     ' Nota: nessuna emissione a video.
VALUE% = INSTR(Instring$, "e")
IF VALUE% = 0 THEN
   PRINT "Grazie. Nella stringa non vi è alcuna 'e'."
ELSE
   PRINT "Grazie. La prima lettera 'e' corrisponde al carattere"
VALUE%
END IF
```

***Consiglio**:* Uno dei vantaggi principati di INPUT$ è dato dal fatto che il suo inserimento non genera alcun tipo di emissione a video dei caratteri ad essa associati. È dunque questo un sistema alternativo di immissione delle istruzioni e funzioni del BASIC.

Innanzitutto verrà visualizzato il contenuto della stringa di richiesta ("Digitare una stringa di 10 caratteri") e in seguito il programma leggerà la stringa immessa:

```
PRINT "Digitare una stringa di 10 caratteri."
Instring$ = INPUT$(10)  ' Nota: nessuna emissione a video.
:
```

Ora verrà ricercata la presenza della prima lettera "e" e, nel caso in cui questa venga reperita, verrà emesso un messaggio che ne indica la sua posizione:

```
PRINT "Digitare una stringa di 10 caratteri."
Instring$ = INPUT$(10)  '      Nota: nessuna emissione a video.
VALUE% = INSTR(Instring$, "e")
IF VALUE% = 0 THEN
   PRINT "Grazie. Nella stringa non vi è alcuna 'e'."
ELSE
   PRINT "Grazie. La prima lettera 'e' corrisponde al carattere"
VALUE%
END IF
```

Tuttavia, la richiesta di uno specifico numero di caratteri (i ritorni di carrello non concludono un'immissione) non è un'operazione così semplice e, anche se è in grado di leggere l'immissione senza visualizzarne il contenuto a video, la funzione INPUT$( ) non potrà restituire i codici del set di caratteri ASCII esteso come, per esempio, i tasti funzione, i tasti frecce, i tasti ALT, ecc.

***Nota**: Per trovare una soluzione a questo problema, si consiglia di analizzare la funzione InKeyNoEcho$( ) presentata più avanti in questo capitolo.*

L'ISTRUZIONE *LINE INPUT*

L'istruzione *LINE INPUT* è molto utile poiché la sua funzione è quella di leggere una stringa di caratteri immessi da tastiera assumendo come fine della stringa la pressione del tasto **INVIO** che genera un ritorno di carrello. Analizzare il listato di seguito riportato:

```
LINE INPUT "Digitare una stringa:";Instring$
PRINT "Grazie. La stringa immessa è composta da"
PRINT ;LEN(Instring$);"caratteri."
```

In questo caso verrà emesso a video il messaggio `Digitare una stringa:`. Si noti che non è stato immesso alcun punto interrogativo e in seguito il programma riceverà la stringa che viene immessa dall'utente. Come si vedrà alla fine di questo capitolo, sulla stringa si potrà operare come meglio si crede.

> ***Consiglio:*** Se si desidera leggere una stringa, l'istruzione LINE INPUT è indubbiamente la soluzione più semplice, poiché e più semplice della funzione INKEY$( ) e più affidabile di INPUT o INPUT$.

Tenere presente che LINE INPUT non consente di gestire la stringa carattere per carattere né è in grado di gestire il set di codici ASCII esteso che vengono, invece, gestiti da *INKEY$*.

INKEY$

Per poter avere un reale controllo sulla stringa è necessario utilizzare la funzione *INKEY$* di cui di seguito viene fornito un esempio:

```
PRINT "Digitare un carattere."

DO
InChar$ = INKEY$
LOOP WHILE InChar$ = ""

PRINT "Grazie. Il carattere immesso è:";InChar$
```

In questo caso il programma rimane in attesa dell'immissione da parte dell'utente di un carattere e poiché *INKEY$* restituisce immediatamente quanto immesso in memoria buffer, verrà attivato il *loop* che verrà continuamente ripetuto finché non si presenterà una situazione simile a quella di seguito riportata:

```
PRINT "Digitare un carattere."

DO
InChar$ = INKEY$
LOOP WHILE InChar$ = ""

PRINT "Grazie. Il carattere immesso è:";InChar$
```

INKEY$ restituisce tre tipi di valori:

1. stringhe di lunghezza zero (stringhe vuote: ""),
2. stringhe di lunghezza uno,
3. stringhe di lunghezza due.

Se la stringa restituita è composta da un solo carattere, verrà rappresentata da un singolo carattere, come, per esempio, *d* oppure *q*.

> ***Consiglio:*** Utilizzare la funzione *LEN( )* per poter determinare la lunghezza della stringa restituita da INKEY$. Se la stringa sarà lunga un solo carattere non vi saranno problemi ma se essa sarà di lunghezza due INKEY$ restituirà un codice del set ASCII esteso.

Se la stringa restituita è di lunghezza due ciò significa che essa rappresenta un *codice ASCII esteso* e il primo carattere della stringa corrisponde al codice ASCII 0; vale a dire, *CHR$(0)*. Il secondo carattere, invece, corrisponde allo *scan code* (codice di scansione) del tasto. Vi è un solo *scan code* per ogni tasto o combinazione di tasti accettabile (come, per esempio, **ALT-K**) e questi potranno essere reperiti nelle tabelle allegate alla documentazione BASIC. Utilizzare le funzioni *RIGHT$( ), LEFT$( )* oppure *MID$( )* per separare il primo e il secondo carattere della stringa che viene restituita.
Sebbene molti programmatori non abbiano ben presente l'uso degli *scan code*, il loro utilizzo consente di incrementare notevolmente l'operatività dei programmi. Verrà ora analizzato come operare con gli *scan code*. Nell'esempio di seguito riportato viene impostata una funzione che indica quale dei tasti ↑ ↓ ← → viene premuto.

GetArrowKey$ – Lettura dei tasti ↑ ↓ ← →

Nell'esempio riportato, viene realizzata una funzione in grado di leggere i tasti ↑ ↓ ← → presenti sul tastierino numerico di una normale tastiera e restituirne il valore in un modo molto semplice da interpretare. Per esempio, nel caso in cui venga premuto il tasto →, il valore restituito potrebbe essere *r*; se viene premuto il tasto ↑,

il valore restituito potrebbe essere *u*. Questa codifica renderà molto facile l'uso dei tasti ↑ ↓ ← → all'interno dei programmi senza dover tenere a mente il loro rispettivo *scan code*. Di seguito viene presentato un ipotetico elenco dei valori che potrebbero essere restituiti da *GetArrowKey$()*.

Valore restituito	*Descrizione*
r	Pressione di →
l	Pressione di ←
u	Pressione di ↑
d	Pressione di ↓
h	Pressione del tasto **HOME**
e	Pressione del tasto **FINE**

La funzione *GetArrowKey$()* potrà essere utilizzata come qualsiasi altra funzione e quando verrà utilizzata essa attenderà che l'utente prema uno dei tasti ↑ ↓ ← → e ne restituirà successivamente la lettera corrispondente. Si provi a passare l'argomento *WarningBeep%* alla funzione *GetArrowKey$()*: se *WarningBeep%* viene impostato a un valore diverso da zero, la funzione *GetArrowKey$()* emetterà un segnale sonoro quando verrà premuto un tasto diverso da ↑ ↓ ← →, **HOME** oppure **FINE**. La funzione è relativamente semplice. Si inizia con la creazione di un loop continuo che rimane in attesa della pressione di un tasto:

```
FUNCTION GetArrowKey$ (WarningBeep%)

   DO
   :
   :
   LOOP WHILE 1
```

Ora sarà possibile accettare l'immissione proveniente da *INKEY$*:

```
FUNCTION GetArrowKey$ (WarningBeep%)

   DO

      DO
         InStr$ = INKEY$
      LOOP WHILE InStr$ = ""
         :
         :
   LOOP WHILE 1
```

Il loop più esterno (DO ... LOOP WHILE 1) viene incessantemente avviato e rimane in attesa fino a quando non verrà premuto un tasto. Dopo che INKEY$ ha restituito il tasto premuto, verrà trasmesso il suo codice ASCII e in seguito verrà controllato

che la sua lunghezza sia pari a due. In caso contrario, e se *WarningBeep%* è diverso da zero, verrà emesso un segnale sonoro.

```
FUNCTION GetArrowKey$ (WarningBeep%)

   DO

      DO
         InStr$ = INKEY$
      LOOP WHILE InStr$ = ""

      Code = ASC(RIGHT$(InStr$, 1))
      IF LEN(InStr$) = 2 THEN
            :
         [check for arrow key]
            :
      ELSE
      IF WarningBeep% THEN BEEP
      END IF
   LOOP WHILE 1
```

Ora sarà necessario controllare quale tasto di spostamento del cursore è stato premuto. Il programma è quello riportato nel Listato 1.3 che utilizza un'istruzione *SELECT CASE* (e lo *scan code* riportato nella documentazione BASIC). In questo programma, se e quando viene premuto uno dei tasti ↑ ↓ ← →, **HOME** o **FINE**, verrà restituito il carattere corretto; in caso contrario si verrà riportati all'inizio e il programma attenderà una nuova pressione di un tasto (dopo aver emesso l'eventuale segnale sonoro).

```
FUNCTION GetArrowKey$ (WarningBeep%)

        DO
            DO
            InStr$ = INKEY$
            LOOP WHILE InStr$ = ""

            Code = ASC(RIGHT$(InStr$, 1))
            IF LEN(InStr$) = 2 THEN
                SELECT CASE Code
                     CASE &H4D
                       GetArrowKey$ = "r"
                       EXIT FUNCTION
                     CASE &H4B
                       GetArrowKey$ = "l"
                       EXIT FUNCTION
```

continua

Listato 1.3. *Funzione GetArrowKey$*

```
                    CASE &H48
                       GetArrowKey$ = "u"

                       EXIT FUNCTION
                    CASE &H50
                       GetArrowKey$ = "d"
                       EXIT FUNCTION
                    CASE &H47
                       GetArrowKey$ = "h"
                       EXIT FUNCTION
                    CASE &H4F
                       GetArrowKey$ = "e"
                       EXIT FUNCTION
                 END SELECT
                 IF WarningBeep% THEN BEEP
              ELSE
                 IF WarningBeep% THEN BEEP
              END IF
           LOOP WHILE 1

END FUNCTION
```

Listato 1.3. *Funzione GetArrowKey$*

Questo programma è uno dei sistemi per utilizzare gli *scan code* che vengono letti da INKEY$. Come si potrà notare la funzione INKEY$ è molto versatile e consentirà di utilizzare valide routine di immissione.
Tuttavia, vi sono occasioni in cui la funzione INKEY$ non è così agevole. Si è già visto che la funzione INPUT$( ) non restituisce alcun tipo di emissione a video ma, purtroppo non è in grado di restituire i codici ASCII estesi.

Consiglio: Se vengono spesso utilizzati gli *scan code* è molto più facile scrivere un breve programma BASIC che sia in grado di emettere la stampa dei dati. Per esempio, per reperire lo *scan code* del tasto →, sarà sufficiente avviare il programma e digitare il tasto richiesto. Il programma stamperà quindi lo *scan code* prelevandolo da INKEY$.

È possibile anche utilizzare una propria versione di INKEY$ che non generi alcuna emissione a video ma che sia in grado di gestire i codici ASCII estesi. Poiché questo processo introduce la routine *INTERRUPT( )* (che verrà utilizzata abbastanza spesso all'interno di questo libro) sarà meglio analizzarlo immediatamente.

InKeyNoEcho$

Si vedrà ora come sviluppare una versione personalizzata di *INKEY$* che agisca esattamente come la versione originale ma che *non* emetta alcun carattere a video. La funzione *InKeyNoEcho$()* non attenderà alcun tipo di immissione, proprio come accade per la funzione INKEY$. Verranno restituiti valori esattamente uguali a quelli che ci si aspetterebbero da INKEY$: una stringa vuota ("") (il che sta a significare che il programma non era in attesa dell'immissione di alcun tipo di carattere), un carattere di lunghezza uno contenente il codice ASCII corrispondente al tasto premuto, un carattere di lunghezza due che prevede l'uso dei codici ASCII estesi.

Lunghezza della stringa restituita	*Descrizione*
0	Stringa vuota (" "). Non è stato immesso alcun carattere dal momento in cui è stata controllata l'immissione da tastiera.
1	Codice ASCII relativo al tasto premuto. Per esempio, se è stato premuto il tasto *q*, la funzione *InKeyNoEcho$* restituirà la lettera *q*.
2	Codice ASCII esteso. Questi tasti non vengono normalmente riportati a video (è il caso dei tasti per lo spostamento del cursore o i tasti funzione). Il primo carattere è *CHR$(0)* e il secondo corrisponde allo *scan code* del tasto (controllare la documentazione BASIC per un elenco completo degli *scan code*).

Ora, poiché nel BASIC non esiste alcuna funzione di questo tipo, sarà necessario inserirla insieme alle altre. Non sarà possibile utilizzare alcun tipo di funzione BASIC di immissione già esistente per creare la funzione personalizzata poiché le funzioni già esistenti generano un'emissione a video oppure sopprimono eventuali *scan code*. Questo sta a significare che è preferibile ricominciare dall'inizio; un sistema per eseguire questa operazione sarà quello di comunicare con il DOS attraverso il sistema degli *interrupt*.

USO DEGLI INTERRUPT BIOS E DOS

Il sistema più valido per incrementare l'operatività del BASIC è quello di utilizzare le risorse che rende disponibile il sistema operativo; sono proprio queste risorse che verranno utilizzate più frequentemente nella programmazione. Ogni *interrupt* è in sostanza un programma "precompilato" e sempre presente in memoria, pronto per essere utilizzato. I comandi che vengono utilizzati a livello DOS (per esempio COPY, TIME, VER, XCOPY oppure FORMAT) fanno tutti uso dello stesso *interrupt* che verrà

utilizzato anche in fase di programmazione BASIC. È un po' come aggiungere interamente un nuovo linguaggio alle possibilità di programmazione disponibili.
Il modo con cui vengono trasmessi e/o ricevuti dati da routine di *interrupt* è basato sull'uso della routine INTERRUPT( ) del BASIC e dei *registri* 80x86. Il microprocessore gestisce i dati in registri a 16 bit e questi potranno essere considerati come variabili predisposte all'interno del computer. I registri che verranno più spesso utilizzati sono denominati *ax*, *bx*, *cx* e *dx* e memorizzeranno i dati della CPU del computer (Figura 1.1).

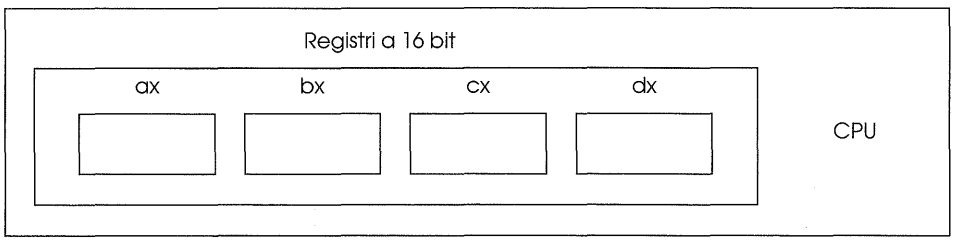

Figura 1.1

Ciascun *interrupt* prenderà in esame il modo in cui sono stati utilizzati uno o più di questi registri ed eseguirà, in conseguenza, le operazioni necessarie. Per un elenco completo di ogni servizio gestito dagli *interrupt* e sul modo in cui verranno utilizzati i registri, si veda l'appendice di questo libro nella quale viene indicato cosa dovrebbe contenere ogni registro, prima di richiamare la routine INTERRUPT( ) oppure INTERRUPTX( ), e che tipo di emissione ci si aspetta di trovare.

Consiglio: INTERRUPT( ) e INTERRUPTX( ) hanno lo stesso valore con la sola eccezione che INTERRUPTX( ) consente di utilizzare un numero leggermente superiore di registri. Utilizzando la routine INTERRUPTX( ) si ha un maggiore controllo sull'operazione.

Poiché ogni registro ha una dimensione di 16 bit è necessario notare che questa dimensione è esattamente uguale INTEGER del BASIC e sarà quindi possibile caricare nei registri valori interi del tipo *53*, *3251* oppure –219 (Figura 1.2).
Per il computer, tuttavia, ogni registro potrà essere considerato come se contenesse 2 byte consentendo così di ripartirli in un registro superiore a 8 bit e un registro inferiore anch'esso di 8 bit. Per esempio, la parte superiore di 8 bit del registro *ax* (oppure *bx*, *cx* ecc.) viene denominata *ah* (*bh*, *ch*, ecc.) e la parte inferiore di 8 bit del registro *ax* (*bx*, *cx*, ecc.) viene denominata *al* (*bl*, *cl*, ecc.) (Figura 1.3).
Accade spesso che sia necessario caricare uno di questi registri a 8 bit, come, per esempio, *ah*, immettendovi un particolare valore per poter utilizzare un *interrupt*.

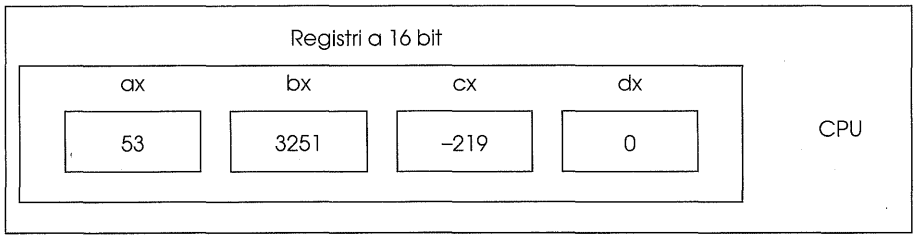

Figura 1.2

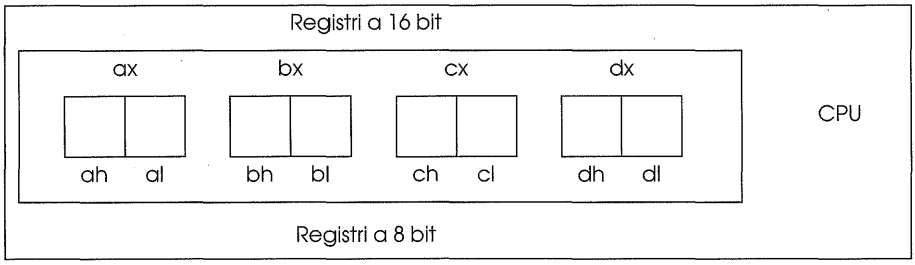

Figura 1.3

Quando si lavora con i registri vengono utilizzati i valori in notazione esadecimale che sono in base 16 dove le cifre vanno da 0 a 9 e in seguito da &HA a &HF.
La notazione esadecimale è molto utile poiché un numero binario a 16 bit — dimensione questa di un intero registro — genera quattro cifre esadecimali. Questo sta a significare che il valore che sarà possibile immettere nei registri 80x86 vanno da 0 a &HFFFF (65535) (figura 1.4).

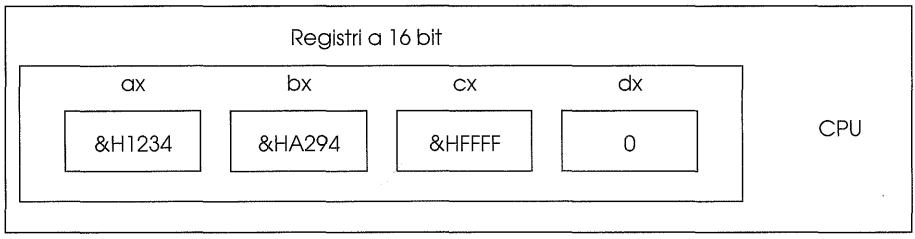

Figura 1.4

Inoltre, 8 bit (corrispondenti a un *byte*) generano esattamente due cifre esadecimali — &H12 oppure &H34 — per cui un byte potrà contenere i valori che vanno da 0 a

&HFF (255). Poiché è possibile dividere un registro come *ax* ripartendolo in *ah* e *al*, il primo byte in una word a 16 bit come *ax* corrisponderà in sostanza alle prime due cifre esadecimali mentre l'ultimo byte corrisponderà alle ultime due. Per esempio, nel caso in cui il registro *ax* contenga &H1234, *ah* conterrà &H12 mentre *al* conterrà &H34 (Figura 1.5).

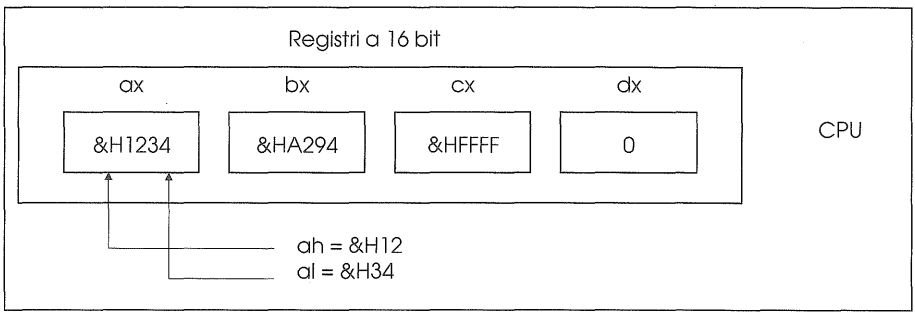

Figura 1.5

Per poter raggiungere con il BASIC i registri 80X86 sarà prima necessario impostare due strutture di dati, *Inregs* e *OutRegs*, assegnando loro il tipo *RegType*. Il tipo viene definito come segue:

```
TYPE RegType
   ax        AS INTEGER
   bx        AS INTEGER
   cx        AS INTEGER
   dx        AS INTEGER
   bp        AS INTEGER
   si        AS INTEGER
   di        AS INTEGER
   flags     AS INTEGER
END TYPE
```

Ora si potrà fare riferimento ai registri *ax* con *InRegs.ax*, a *bx* con *InRegs.bx*, eccetera. Si osservi che sarà necessario riservare uno spazio per quattro nuovi registri (*bp, si, di* e *flags*) (Figura 1.6).

***Nota**: I byte superiore e inferiore di questi nuovi registri non potranno essere indirizzati separatamente: per esempio, si non potrà essere suddiviso in sh e si.*

Con la sola eccezione dei registri *flags*, questi nuovi registri non verranno presi in considerazione fino a quando si inizierà a trattare il linguaggio assembly. Il registro *bp* viene di solito utilizzato per manipolare i dati sotto forma di *stack* e verrà trattato nel Capitolo 11. I registri *bp* e *si* vengono normalmente utilizzati da 80X86 per poter

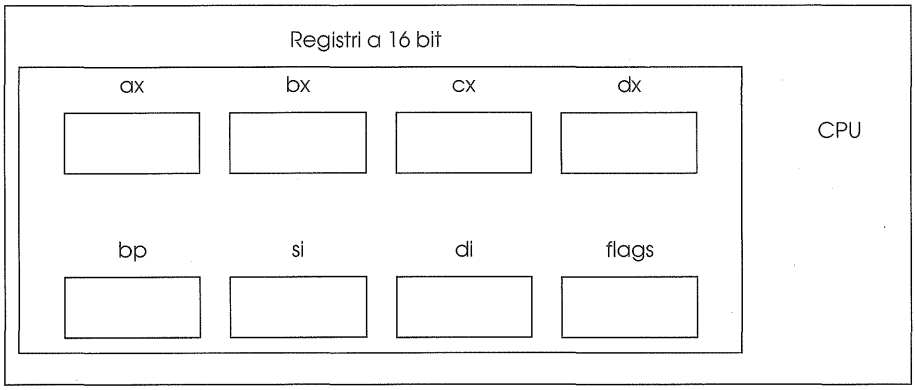

Figura 1.6

manipolare le stringhe e verranno presi dettagliatamente in esame nel Capitolo 10. In questo capitolo, invece, vengono trattati i registri *flag*.
Vi sono 9 *flag* comuni per tutti i processori 80x86 (che prevedono 13 *flag*). Questi *flag* normalmente riportano lo stato delle operazioni matematiche. Per esempio, vi è un *flag* zero che viene impostato nel caso in cui il risultato di un'operazione sia pari a zero. Il *flag* di riporto viene normalmente impostato se l'ultima operazione eseguita prevede la presenza di un riporto. I bit di ordine inferiore di *InRegs.flags* contengono i nove più comuni *flag* 80x86 che vengono di seguito elencati, bit per bit.

Bit InRegs.flags	*Flag*
11	Flag di overflow
10	Flag di direzione
9	Flag abilitazione interrupt
8	Flag trap
7	Flag segno
6	Flag zero
4	Flag ausiliario riporto
2	Flag di parità
0	Flag di riporto

Ora che è stato definito *RegType*, si è liberi di utilizzare *INTERRUPT( )*. Per esempio, per generare l'emissione a video di un carattere, prima sarà necessario controllare l'appendice per verificare che quello che si desidera utilizzare è l'*interrupt* &H21, servizio 2. Di seguito viene mostrato il contenuto di INT &H21, servizio 2:

Input
ah=2
dl=Codice ASCII del carattere

Poiché l'*interrupt* &H21 è in grado di eseguire diverse operazioni, esso è diviso in diversi servizi; il servizio 2 è quello che consente l'emissione a video (si veda l'Appendice per un elenco completo dei servizi di *INTERRUPT &H21*). Per selezionare un servizio di un'*interrupt*, caricare il numero relativo nel registro *ah* (che corrisponde al byte alto del registro *ax*). Poiché si potrà lavorare solamente con valori interi, sarà necessario caricare *2* nel primo byte di *ax* (le prime due cifre di un numero di quattro cifre in notazione esadecimale occupano il primo byte):

```
DIM InRegs AS RegType, OutRegs AS regType
InRegs.ax = &H0200
:
```

Sarà ora necessario caricare il codice ASCII del carattere nel registro *dl*, corrispondente al byte basso del registro *dx*. Si presupponga di dover generare una "*A*".

```
DIM InRegs AS RegType, OutRegs AS regType
InRegs.ax = &H0200
InRegs.dx = ASC("A")
:
```

Successivamente, verrà immessa la chiamata a INTERRUPT( ) che eseguirà l'*interrupt*. Sarà ora necessario passare il numero dell'*interrupt* che si intende utilizzare (&H21) e la variabile di tipo *RegType* che contiene i valori che si desidera immettere nei registri precedentemente definiti *InRegs*. I servizi dell'*interrupt* dovranno anch'essi generare l'emissione di valori e questi verranno trasferiti in *OutRegs*.

```
DIM InRegs AS RegType, OutRegs AS regType
InRegs.ax = &H0200
InRegs.dx = ASC("A")
CALL INTERRUPT(&H21, InRegs, OutRegs)
   :
```

E per ora è tutto: il carattere *A* verrà visualizzato a video nella posizione corrente del cursore. Ora sarà necessario interfacciare il BASIC con il DOS utilizzando la routine INTERRUPT( ).

***Consiglio**:* Tutti i programmi che utilizzano INTERRUPT( ) oppure INTERRUPTX( ) dovranno essere caricati con l'opzione */L* nel caso in cui si stia utilizzando QuickBASIC (QB.EXE oppure QBX.EXE) servendosi rispettivamente di una delle seguenti righe di comando:

```
QB PROG /L
QBX PROG /L
```

UN SERVIZIO DOS PER LA TASTIERA

Con la funzione *InKeyNoEcho$()* si accederà a un servizio DOS, e più precisamente all'interrupt &H21, servizio 6. Questo servizio esegue esattamente quello che ci si aspetta, ovvero, legge quanto immesso da tastiera ma non ne emette a video il risultato. Di seguito viene riportato il contenuto di INT &H21, servizio 6 console I/O senza echo (si veda anche l'Appendice).

Input		**Output**
ah = 6		
dl = &HFF	→	Il *flag* zero viene impostato a zero nel caso in cui non sia pronto alcun carattere. In caso contrario in *al* viene immesso il codice ASCII del carattere digitato.
dl < &HFF	→	Emette a video il codice ASCII contenuto in *dl*.

Per poter utilizzare questo servizio, sarà necessario immettere 6 in *ah* e &HFF in *dl*. Se verrà immesso in *dl* un altro valore diverso da &HFF, questo servizio verrà trasformato in servizio di stampa. Il DOS tratterà il valore presente in *dl* come un semplice codice ASCII e ne emetterà il risultato a video. In questo caso, tuttavia, poiché si desidera ricevere il risultato di un'immissione, è necessario immettere &HFF in *dl*. La chiamata a INTERRUPT( ) dovrebbe essere come quella di seguito riportata:

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.ax = &H0600
InRegs.dx = &HFF
CALL INTERRUPT(&H21, InRegs, OutRegs)
   :
```

Questo servizio restituisce diversi valori. Se il *flag* zero di 80x86 è stato impostato su restituzione allora non vi sarà alcuna attesa del carattere da leggere. Il *flag* zero è uno dei nove *flag* interni di 80x86; dall'elenco precedentemente presentato, si può notare che il *flag* zero è il bit numero 6 di *OutRegs.flags*.

Bit di OutRegs.flags	*Flag*
11	Flag di overflow
10	Flag di direzione
9	Flag abilitazione interrupt
8	Flag Trap
7	Flag segno
6	Flag zero
4	Flag ausiliario di riporto
2	Flag parità
0	Flag di riporto

Questo sta a significare che sarà sempre possibile controllare se c'è in attesa un carattere:

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.ax = &H0600
InRegs.dx = &HFF
CALL INTERRUPT(&H21, InRegs, OutRegs)

REM Nessun carattere pronto se è stato impostato il flag zero

IF (OutRegs.flags AND 2^6) THEN
   InKeyNoEcho$ = ""
   :
   :
```

Nel caso in cui non vi sia nessun carattere in attesa (per esempio quando è stato impostato il bit del flag zero) allora *OutRegs.flags AND 2^6* sarà diverso da zero e si assegnerà una stringa vuota a *InKeyNoEcho$*, proprio come ci si aspettava di ottenere da *INKEY$*.
D'altra parte, se vi è in attesa un carattere, sarà necessario esaminarne il valore che verrà restituito nel registro *al* (gli 8 bit di ordine inferiore del registro *OutRegs.ax*). Se il valore è diverso da zero, allora questo risulterà essere il codice ASCII del tasto premuto e sarà necessario trasformarlo in un carattere di stringa BASIC prima di restituirne il risultato.

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.ax = &H0600
InRegs.dx = &HFF
CALL INTERRUPT(&H21, InRegs, OutRegs)

REM Nessun carattere pronto se è stato impostato il flag zero

IF (OutRegs.flags AND 2^6) THEN
   InKeyNoEcho$ = ""
ELSE
   IF (OutRegs.ax AND &HFF)
      InKeyNoEcho$ = CHR$(OutRegs.ax AND &HFF)
   :
   :
```

Ancora una volta questo è il risultato che ci si aspettava da INKEY$. Infine, se il codice ASCII contenuto in *al* è pari a 0 dopo la chiamata di questo servizio, questo sta a indicare che il tasto premuto corrisponde a un codice ASCII di tipo esteso; in questo caso sarà necessario effettuare una successiva chiamata al servizio 6 in modo da poter ottenere lo *scan code* del tasto. A questo punto si dovrà restituire una stringa contenente due caratteri (CHR$(0)) e lo *scan code* del tasto:

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.ax = &H0600
```

```
InRegs.dx = &HFF
CALL INTERRUPT(&H21, InRegs, OutRegs)

REM Nessun carattere pronto se è stato impostato il flag zero

IF (OutRegs.flags AND 2^6) THEN
   InKeyNoEcho$ = ""
ELSE
   IF (OutRegs.ax AND &HFF)
      InKeyNoEcho$ = CHR$(OutRegs.ax AND &HFF)
   ELSE       ' E' necessaria un'ulteriore chiamata al servizio 6
      InRegs.ax = &H0600
      InRegs.dx = &HFF
      CALL INTERRUPT(&H21, InRegs, OutRegs)
      InKeyNoEcho$ = CHR$(0) + CHR$(OutRegs.ax AND &HFF)
   END IF
END IF
```

E questo è proprio quello che ci si aspettava da INKEY$ nel caso in cui si richiedeva la restituzione di un codice ASCII esteso. In questo modo, la funzione *InKeyNoEcho$( )* non fa altro che emulare quanto si sarebbe ottenuto da INKEY$, con la sola eccezione che non viene visualizzato nulla a video man mano che l'utente preme i vari tasti.

> ***Consiglio***: Si suggerisce di utilizzare la funzione *InKeyNoEcho$( )* quando ci si trova in modalità grafica o quando si desidera digitare una parola d'ordine, poiché quanto digitato non viene restituito a video.

Il listato 1.4 mostra la funzione completa.

```
TYPE RegType
        ax      AS INTEGER
        bx      AS INTEGER
        cx      AS INTEGER
        dx      AS INTEGER
        bp      AS INTEGER
        si      AS INTEGER
        di      AS INTEGER
        flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)
```

continua

Listato 1.4 *La funzione InKeyNoEcho$( )*

```
        PRINT "Digitare un carattere: "
        DO
        TheKey$ = InKeyNoEcho$
        LOOP WHILE TheKey$ = ""
        PRINT "Il carattere digitato è: ",TheKey$

END

FUNCTION InKeyNoEcho$

   DIM InRegs AS RegType, OutRegs AS RegType
   InRegs.ax = &H0600
   InRegs.dx = &HFF

   CALL INTERRUPT(&H21, InRegs, OutRegs)

   REM Nessun carattere pronto se è stato impostato il flag zero

   IF (OutRegs.flags AND 2^6) THEN
       InKeyNoEcho$ = ""
   ELSE
       IF (OutRegs.ax AND &HFF)  0 THEN
           InKeyNoEcho$ = CHR$(OutRegs.ax AND &HFF)
       ELSE    'E' necessaria un'ulteriore chiamata al servizio 6
           InRegs.ax = &H0600
           InRegs.dx = &HFF
           CALL INTERRUPT(&H21, InRegs, OutRegs)
           InKeyNoEcho$ = CHR$(0) + CHR$(OutRegs.ax AND &HFF)
      END IF
   END IF

END FUNCTION
```

Listato 1.4 *La funzione InKeyNoEcho$( )*

Con *InKeyNoEcho$( )* si è compiuto un ulteriore passo in avanti: si è fatto uso di un servizio DOS e si sono utilizzate alcune istruzioni e funzioni BASIC per le immissioni di dati da tastiera (INPUT, INPUT$, LINE INPUT e INKEY$).
Il passo successivo sarà quello di consolidare queste funzioni elementari. Nei programmi di tipo professionale, le routine di immissione devono assolutamente avere un aspetto ordinato e facilmente leggibile e si vedrà ora come realizzare e sviluppare un programma che sia in grado di leggere un valore intero.

SCRITTURA DI UNA FUNZIONE DI IMMISSIONE

Si vedrà ora come dovrebbe apparire la realizzazione professionale di una funzione di immissione che sia in grado di leggere i valori interi che vengono immessi da tastiera. Assegneremo alla funzione il nome di *GetInteger%( )* e ad essa verrà assegnata la numerazione delle righe in modo da evitare possibili errori e la presentazione di messaggio del tipo `Redo from start`. La presentazione di un messaggio di questo tipo significa la presenza di una condizione di overflow oppure di caratteri non ammessi. Se per esempio viene immesso un numero non appropriato, potremo indicare alla funzione *GetInteger%( )* di cancellare semplicemente il numero scorretto e di riavviare nuovamente il programma spostando il cursore alla posizione in cui esso si trovava quando è stata invocata la funzione *GetInteger%( )*. Se il codice è stato immesso in modo ordinato, la funzione non influenzerà in alcun modo l'aspetto globale del programma. Inoltre, come già fatto per la funzione *GetArrowKey$( )*, potremmo utilizzare anche un parametro a cui assegneremo il nome *WarningBeep%*; nel caso in cui questo valore sia diverso da zero, la funzione *GetInteger%( )* genererà un segnale sonoro se viene immesso un numero non intero. Si passerà ora alla realizzazione vera e propria della funzione. Innanzitutto sarà necessario salvare la posizione corrente del cursore e in seguito impostare un ciclo del tipo *DO ... LOOP WHILE 1* che venga continuamente eseguito (l'unico modo per abbandonare *GetInteger%( )* è quello di immettere un valore intero).

```
FUNCTION GetInteger% (WarningBeep%)
   CursorRow% = CSRLIN
   CursorCol% = POS(0)
   DO
      :
      :
   LOOP WHILE 1
```

All'inizio del ciclo viene immessa una stringa alla quale è stato assegnato il nome *InString$*. Questo ciclo continuerà ad essere eseguito fino a quando il programma non sarà in grado di decodificare quanto immesso come valore intero:

```
FUNCTION GetInteger% (WarningBeep%)
   CursorRow% = CSRLIN
   CursorCol% = POS(0)
   DO
      LINE INPUT InString$
      :
      :
   LOOP WHILE 1
```

Ora si potrà effettuare un controllo preliminare sulla dimensione del numero osservandone la sua lunghezza; se è superiore a sei caratteri si dovrà iniziare nuovamente da capo (la lunghezza massima di un valore intero deve essere di sei caratteri — cinque cifre e un segno). La subroutine *StarOver*riposiziona il cursore alle coordinate di origine cancellando la stringa immessa; questa subroutine viene eseguita solo nel caso i cui la stringa ha una lunghezza superiore a 6 cartteri.

```
FUNCTION GetInteger% (WarningBeep%)

   CursorRow% = CSRLIN
   CursorCol% = POS(0)
   DO
      LINE INPUT InString$

      IF LEN(InString$) < = 6 THEN
      :
      [Il numero dovrebbe essere corretto]
      :
      ELSE
         GOSUB StartOver
      END IF        ' IF LEN(InString$) < = 6 ...

   LOOP WHILE 1

StartOver:
   IF WarningBeep% THEN BEEP
   LOCATE CursorRow%, CursorCol%
   PRINT SPACES(LEN(InString$));
   LOCATE CursorRow%, CursorCol%
   RETURN
```

Ora sarà necessario avviare un *loop* su ogni carattere presente nella stringa di immissione *InString$*. Se il carattere è compreso tra 0 e 9 questo verrà aggiunto al totale che risulterà nel valore intero finale. I segni + e – verranno ritenuti validi *solo* se compaiono come primo carattere della stringa. In caso contrario o nel caso in cui il risultato sia un carattere scorretto, sarà necessario svuotare la stringa che viene riportata a video e reimpostare la posizione del cursore alla sua posizione originale. Di seguito viene indicato come controllare ciascun carattere:

```
FUNCTION GetInteger% (WarningBeep%)

   CursorRow% = CSRLIN
   CursorCol% = POS(0)
   DO
      LINE INPUT InString$

      IF LEN(InString$) < = 6 THEN
         FOR i = 1 TO LEN(InString$)
```

```
            Char$ = MID$(InString$, i, 1)
            IF Char$ > = "0" AND Char$ < = "9" THEN
               :
               Somma la cifra al totale corrente
            ELSE
               :
               Controlla la presenza di + o - in prima posizione
               :
            END IF
         NEXT i
      ELSE
         GOSUB StartOver
      END IF        ' IF LEN(InString$) < = 6 ...

   LOOP WHILE 1

StartOver:
   IF WarningBeep% THEN BEEP
   LOCATE CursorRow%, CursorCol%
   PRINT SPACES(LEN(InString$));
   LOCATE CursorRow%, CursorCol%
   RETURN
```

Se il carattere risultante è accettabile questo dovrà essere aggiunto al totale corrente, controllarne l'eventuale generazione di *overflow* (il totale corrente dovrebbe corrispondere a un intero di tipo LONG per evitare problemi con il BASIC) e, in questo caso, reiniziare il processo. Sarà ora necessario immettere il totale corrente in una variabile a cui si assegnerà il nome di *SUM&* come di seguito indicato:

```
FUNCTION GetInteger% (WarningBeep%)

   CursorRow% = CSRLIN
   CursorCol% = POS(0)
   DO
      LINE INPUT InString$
      NegFlag% = 0
      Sum& = 0
      IF LEN(InString$) < = 6 THEN
         FOR i = 1 TO LEN(InString$)
            Char$ = MID$(InString$, i, 1)
            IF Char$ > = "0" AND Char$ < = "9" THEN
               SUM& = SUM& + (ASC(Char$)-ASC("0"))*10^(LEN
               (InString$)-1)
               :
               :
            END IF
         NEXT i
      ELSE
```

```
      GOSUB StartOver
   END IF        ' IF LEN(InString$) < = 6 ...

LOOP WHILE 1

StartOver:
   IF WarningBeep% THEN BEEP
   LOCATE CursorRow%, CursorCol%
   PRINT SPACES(LEN(InString$));
   LOCATE CursorRow%, CursorCol%
   RETURN
```

> ***Consiglio***: È possibile convertire le lettere in numeri sottraendo il codice ASCII corrispondente a 0. Per esempio, ASC("0") – ASC("0") è pari a 0; ASC("1") – ASC("0") è pari a 1, ecc.

Per ogni cifra sarà necessario controllare il contenuto di SUM& e nel caso in cui questo superi i limiti imposti dai numeri di tipo INTEGER, verrà emesso un segnale sonoro (se richiesto) e il processo ricomincerà:

```
FUNCTION GetInteger% (WarningBeep%)

   CursorRow% = CSRLIN
   CursorCol% = POS(0)
   DO
      LINE INPUT InString$
      NegFlag% = 0
      Sum& = 0
      IF LEN(InString$) < = 6 THEN
         FOR i = 1 TO LEN(InString$)
            Char$ = MID$(InString$, i, 1)
            IF Char$ > = "0" AND Char$ < = "9" THEN
               SUM& = SUM& + (ASC(Char$)-ASC("0"))*10^(LEN
               (InString$)-1)
               IF SUM& > 32767 THEN ' Overflow?
                  GOSUB StartOver
                  EXIT FOR
               END IF
               :
               :
            END IF
         NEXT i
      ELSE
         GOSUB StartOver
      END IF        ' IF LEN(InString$) < = 6 ...
   LOOP WHILE 1
```

```
StartOver:
   IF WarningBeep% THEN BEEP
   LOCATE CursorRow%, CursorCol%
   PRINT SPACES(LEN(InString$));
   LOCATE CursorRow%, CursorCol%
   RETURN
```

In caso contrario, il controllo verrà effettuato se è stata raggiunta la fine della stringa di immissione. In questo caso il processo è terminato e *GetInteger%* viene impostato a *SUM&* e si esce dalla funzione:

```
FUNCTION GetInteger% (WarningBeep%)

   CursorRow% = CSRLIN
   CursorCol% = POS(0)
   DO
      LINE INPUT InString$
      NegFlag% = 0
      Sum& = 0
      IF LEN(InString$) < = 6 THEN
         FOR i = 1 TO LEN(InString$)
            Char$ = MID$(InString$, i, 1)
            IF Char$ > = "0" AND Char$ < = "9" THEN
               SUM& = SUM& + (ASC(Char$)-ASC("0"))*10^(LEN
               (InString$)-1)
               IF SUM& > 32767 THEN ' Overflow?
                  GOSUB StartOver
                  EXIT FOR
               END IF
               IF i = LEN(InString$) THEN
                  IF NegFlag% = SUM&
                  EXIT FUNCTION
               END IF
            ELSE
               :
               :
            END IF
         NEXT i
      ELSE
         GOSUB StartOver
      END IF        ' IF LEN(InString$) < = 6 ...

   LOOP WHILE 1
StartOver:
   IF WarningBeep% THEN BEEP
   LOCATE CursorRow%, CursorCol%
   PRINT SPACES(LEN(InString$));
   LOCATE CursorRow%, CursorCol%
   RETURN
```

> ***Consiglio***: Il BASIC consente di assegnare variabili di due tipi diversi come nel caso dell'istruzione *GetInteger% = SUM&*. In questo caso il valore *SUM&* viene troncato (vengono persi i bit di livello superiore) in modo che esso possa essere contenuto nel valore intero definito da *GetInteger%*.

Fino ad ora sono stati utilizzati solo caratteri corrispondenti a cifre ASCII e perciò sarà ora necessario controllare che il carattere corrente sia un + oppure un –. Qualsiasi altro carattere genera la ripetizione della procedura. I caratteri corrispondenti al segno verranno ammessi in valori interi solo nel caso in cui essi appaiano come *primo* carattere. Di seguito viene riportato come effettuare questo controllo:

```
FUNCTION GetInteger% (WarningBeep%)

   CursorRow% = CSRLIN
   CursorCol% = POS(0)
   DO
      LINE INPUT InString$
      NegFlag% = 0
      Sum& = 0
      IF LEN(InString$) < = 6 THEN
         FOR i = 1 TO LEN(InString$)
            Char$ = MID$(InString$, i, 1)
            IF Char$ > = "0" AND Char$ < = "9" THEN
               SUM& = SUM& + (ASC(Char$)-ASC("0"))*10^(LEN
               (InString$)-1)
               IF SUM& > 32767 THEN ' Overflow?
                  GOSUB StartOver
                  EXIT FOR
               END IF
               IF i = LEN(InString$) THEN
                  IF NegFlag% = SUM&
                  EXIT FUNCTION
               END IF
            ELSE
               IF (Char$ = "-" OR Char$ = "+") AND i = 1 THEN
               :
               [Il carattere è un segno accettabile]
               :
            END IF
         NEXT i
      ELSE
         GOSUB StartOver
      END IF        ' IF LEN(InString$) < = 6 ...

   LOOP WHILE 1
```

```
StartOver:
   IF WarningBeep% THEN BEEP
   LOCATE CursorRow%, CursorCol%
   PRINT SPACES(LEN(InString$));
   LOCATE CursorRow%, CursorCol%
   RETURN
```

Se il segno è un –, allora si noterà che l'impostazione del *flag* negativo (*NegFlag%*) viene impostato a 1 in modo che esso possa in seguito restituire –*SUM&*. Si noti inoltre che ci si deve accertare, anche se il segno corrisponde al primo carattere della stringa, della presenza di successivi caratteri (ossia le cifre). D'altronde, anche i soli segni + o – risulteranno caratteri accettabili. Di seguito viene riportato il sistema per effettuare questo controllo, che evidenzia, se presente, un segno negativo.

```
FUNCTION GetInteger% (WarningBeep%)

   CursorRow% = CSRLIN
   CursorCol% = POS(0)
   DO
      LINE INPUT InString$
      NegFlag% = 0
      Sum& = 0
      IF LEN(InString$) < = 6 THEN
         FOR i = 1 TO LEN(InString$)
            Char$ = MID$(InString$, i, 1)
            IF Char$ > = "0" AND Char$ < = "9" THEN
               SUM& = SUM& + (ASC(Char$)-ASC("0"))*10^(LEN
               (InString$)-1)
               IF SUM& > 32767 THEN ' Overflow?
                  GOSUB StartOver
                  EXIT FOR
               END IF
               IF i = LEN(InString$) THEN
                  IF NegFlag% = SUM&
                  EXIT FUNCTION
               END IF
            ELSE
               IF (Char$ = "-" OR Char$ = "+") AND i = 1 THEN
                  IF LEN(InString$) = 1 THEN
                     GOSUB StartOver
                     EXIT FOR
                  END IF
                  IF Char$ = "-" THEN NegFlag% = 1
               ELSE
                  :
               Il carattere non è compreso tra "0" e "9"
               né è "+" o "-" -- ricomincia dall'inizio.
               :
```

```
            END IF
         END IF
      NEXT i
   ELSE
      GOSUB StartOver
   END IF        ' IF LEN(InString$) < = 6 ...

LOOP WHILE 1

StartOver:
   IF WarningBeep% THEN BEEP
   LOCATE CursorRow%, CursorCol%
   PRINT SPACES(LEN(InString$));
   LOCATE CursorRow%, CursorCol%
   RETURN
```

Ora che si è terminato il controllo dei caratteri non accettabili (caratteri che non siano compresi tra 0 e 9 oppure che non siano un segno valido) si potrà richiedere, in caso di carattere errato, l'emissione di un segnale sonoro e una nuova ripetizione della procedura:

```
FUNCTION GetInteger% (WarningBeep%)

   CursorRow% = CSRLIN
   CursorCol% = POS(0)
   DO
      LINE INPUT InString$
      NegFlag% = 0
      Sum& = =
      IF LEN(InString$) < = 6 THEN
         FOR i = 1 TO LEN(InString$)
            Char$ = MID$(InString$, i, 1)
            IF Char$ > = "0" AND Char$ < = "9" THEN
               SUM& = SUM& + (ASC(Char$)-ASC("0"))*10^(LEN
               (InString$)-1)
               IF SUM& > 32767 THEN ' Overflow?
                  GOSUB StartOver
                  EXIT FOR
               END IF
               IF i = LEN(InString$) THEN
                  IF NegFlag% = SUM&
                  EXIT FUNCTION
               END IF
            ELSE
               IF (Char$ = "-" OR Char$ = "+") AND i = 1 THEN
                  IF LEN(InString$) = 1 THEN
                     GOSUB StartOver
                     EXIT FOR
                  END IF
```

```
                IF Char$ = "-" THEN NegFlag% = 1
             ELSE
                GOSUB StartOver
                EXIT FOR
             END IF
          END IF
       NEXT i
    ELSE
       GOSUB StartOver
    END IF        ' IF LEN(InString$) < = 6 ...

  LOOP WHILE 1

StartOver:
  IF WarningBeep% THEN BEEP
  LOCATE CursorRow%, CursorCol%
  PRINT SPACES(LEN(InString$));
  LOCATE CursorRow%, CursorCol%
  RETURN
```

La funzione è ora completa e come si è potuto notare, la creazione di in'immissione di tipo *INTEGER* non è sempre una facile operazione.
Il listato 1.5 riporta il contenuto dell'intera funzione.

> ***Consiglio***: Utilizzare la funzione *GetInteger%()* in sostituzione a INKEY$ oppure LINE INPUT quando viene richiesta l'immissione di un dato numerico, come nel caso di un foglio elettronico, di un'applicazione di calcolo oppure un programma di contabilità.

```
DECLARE FUNCTION GetInteger% (WarningBeep%)

FUNCTION GetInteger% (WarningBeep%)

    CursorRow% = CSRLIN
    CursorCol% = POS(0)
    DO
        LINE INPUT InString$

        NegFlag% = 0
        SUM& = 0
```

continua

Listato 1.5 *La funzione GetInteger%.*

```
        IF LEN(InString$) <= 6 THEN
            FOR i = 1 TO LEN(InString$)
                Char$ = MID$(InString$, i, 1)
                IF Char$ >= "0" AND Char$ <= "9" THEN
                    SUM& = SUM& + (ASC(Char$) - ASC("0"))
                           * 10 ^ (LEN(InString$) - i)
                    IF SUM& > 32767 THEN       'Overflow?
                        GOSUB StartOver
                        EXIT FOR
                    END IF
                    IF i = LEN(InString$) THEN
                        IF NegFlag% THEN SUM& = -SUM&
                        GetInteger% = SUM&
                        EXIT FUNCTION
                    END IF
                ELSE
                    IF (Char$ = "-" OR Char$ = "+")
                         AND i = 1 THEN
                        IF LEN(InString$) = 1 THEN
                           GOSUB StartOver
                           EXIT FOR
                        END IF
                        IF Char$ = "-" THEN NegFlag% = 1
                    ELSE
                        GOSUB StartOver
                        EXIT FOR
                    END IF    'IF segno ammesso
                END IF   'IF Char$ >= "0" AND Char$ <= "9"...
            NEXT i
        ELSE
                GOSUB StartOver
        END IF  'IF LEN(InString$) <= 6...

    LOOP WHILE 1


StartOver:

        IF WarningBeep% THEN BEEP
        LOCATE CursorRow%, CursorCol%
        PRINT SPACE$(LEN(InString$));
        LOCATE CursorRow%, CursorCol%
        RETURN

        END FUNCTION
```

Listato 1.5 *La funzione GetInteger%*

Si è così fatto un ulteriore passo in avanti realizzando una routine di immissione utilizzando le funzioni basilari di BASIC. Il passo successivo sarà quello di operare con routine di immissione più complicate; per esempio, si potrà suddividere la routine in codici più brevi e maneggevoli.

INTERPRETAZIONE DELLE IMMISSIONI DA TASTIERA

L'esempio che verrà presentato per la suddivisione di una routine è relativamente semplice. Verrà creato un semplice programma che utilizza la notazione polacca (notazioni senza parentesi). Il programma sarà in grado di accettare e calcolare espressioni di lunghezza variabile, come quelli di seguito riportati:

2 3 + 4 – [Risultato: 1]
3.1 2.1 – 10 * 5 + [Risultato: 6]

In altre parole, nel primo esempio, *2 3 + 4 –*, la cifra 2 è stata posizionata nello *stack* del calcolatore e in seguito ad essa viene sommato il *3*, il cui risultato dà *5*. Dal risultato ottenuto viene in seguito sottratto *4* generando un risultato finale pari a *1*.
Si vedrà ora come eseguire questa procedura. I codici che verranno presentati non sempre saranno graficamente piacevoli poiché in un programma reale i vari elementi dovranno essere in grado di gestire anche gli spazi, diversi tipi di delimitatore, caratteri immessi in lettere maiuscole e minuscole, spazi di riempimento prima e dopo le cifre e qualsiasi altro tipo di numero di tipo più complesso. Tuttavia, l'uso di questo tipo di suddivisione in codici richiederebbe esempi troppo lunghi e lo scopo di questo esempio è quello di dimostrare unicamente l'uso della *frammentazione* del programma che consente l'analisi di frammenti elementari facilmente leggibili dell'intero programma.

Consiglio: Dopo aver analizzato attentamente le routine del programma, questi potranno essere successivamente utilizzati in tutti i programmi che dovranno leggere i dati di immissione.

Innanzitutto sarà necessario immettere l'espressione che sia in grado di avviare il calcolo da tastiera e ad essa verrà assegnato il nome *CurrentString$*.

```
LINE INPUT "Espressione da calcolare:"; CurrentString$
:
:
```

Sarà ora necessario inserire alcuni controlli minimi di errore per accertarsi che *CurrentString$* sia per lo meno lunga cinque caratteri, corrispondenti al minimo assoluto (per esempio: *2 3* +).

```
LINE INPUT "Espressione da calcolare:"; CurrentString$
IF LEN(CurrentString$) >= 5 THEN      ' minimo 5 caratteri
:
:
END IF
```

Successivamente si potrà richiamare la funzione *NextTerm$( )* che esegue la prima analisi. Per esempio, nel caso in cui *CurrentString$* sia uguale a *2 3* + *4*–, la funzione *NextTerm$(CurrentString$)* dovrebbe restituire 2 e *CurrentString$* verrà troncato a *3* + *4* –. Se *CurrentString$* era, per esempio, *3.1 2.1* – *10* * *5* + la funzione *NextTerm$(CurrentString$)* dovrebbe restituire *3.1* e *CurrentString$* dovrebbe generare un troncamento a *2.1* – *10* * *5* +.

Ora il primo termine (*2* nell'esempio 2 3 +) che verrà caricato nella variabile a cui si assegnerà il nome *RPStack!* e la cui funzione è quella di inserire il valore nello stack del calcolatore. La stringa restituita da *NextTerm$( )* potrà essere convertita in un valore numerico con la funzione *VAL( )* del BASIC.

```
DECLARE FUNCTION NextTerm$ (StringToParse$)

LINE INPUT "Espressione da calcolare:"; CurrentString$

IF LEN(CurrentString$) >= 5 THEN      ' minimo 5 caratteri
   RPStack! = VAL(NextTerm$(CurrentString$))   ' primo termine
:
:
END IF
PRINT "Risultato:", RPStack!
```

Ora sarà necessario ottenere il termine successivo seguito da un operatore ed eseguire l'operazione. Si dovrà proseguire fintanto che in *CurrentString$* saranno contenuti caratteri perciò è necessario impostare un *loop* WHILE che verrà continuamente eseguito data la condizione *CurrentString$* <> "".

```
DECLARE FUNCTION NextTerm$ (StringToParse$)

LINE INPUT "Espressione da calcolare:"; CurrentString$

IF LEN(CurrentString$) >= 5 THEN      ' minimo 5 caratteri
   RPStack! = VAL(NextTerm$(CurrentString$)) ' primo termine
   DO
      NewTerm! = VAL(NextTerm$(CurrentString$))
```

```
      Operator$ = NextTerm$(CurrentString$)
         :
   LOOP WHILE CurrentString$ <> ""
END IF
```

Le operazioni potranno ora essere eseguite utilizzando un'istruzione *SELECT CASE.*

```
DECLARE FUNCTION NextTerm$ (StringToParse$)

LINE INPUT "Espressione da calcolare:"; CurrentString$

IF LEN(CurrentString$) >= 5 THEN      ' minimo 5 caratteri
   RPStack! = VAL(NextTerm$(CurrentString$))' primo termine
   DO
      NewTerm! = VAL(NextTerm$(CurrentString$))
      Operator$ = NextTerm$(CurrentString$)
      SELECT CASE Operator$
         CASE "+"
            RPStack! = RPStack! + NewTerm!
         CASE "-"
            RPStack! = RPStack! - NewTerm!
         CASE "*"
            RPStack! = RPStack! * NewTerm!
         CASE "/"
            RPStack! = RPStack! / NewTerm!
         CASE ELSE
            PRINT "Operatore non corretto:"; Operator$
            END
      END SELECT
   LOOP WHILE CurrentString$ <> ""
END IF
```

> ***Consiglio:*** L'istruzione *SELECT CASE* del BASIC deve essere utilizzata per sostituire una "scala" oppure una serie di istruzioni del tipo *IF ... THEN ... ELSE.* Si suggerisce di utilizzare SELECT CASE ogni volta che è possibile sostituirlo alle istruzioni IF: il codice verrà eseguito più rapidamente.

Alla fine di questo ciclo su *CurrentString$* è praticamente conclusa la stringa di immissione e il risultato viene mantenuto in *RPStack!.* Il risultato, a questo punto dovrebbe essere:

```
DECLARE FUNCTION NextTerm$ (StringToParse$)

LINE INPUT "Espressione da calcolare:"; CurrentString$

IF LEN(CurrentString$) >= 5 THEN      ' minimo 5 caratteri
   RPStack! = VAL(NextTerm$(CurrentString$))' primo termine
```

```
    DO
       NewTerm! = VAL(NextTerm$(CurrentString$))
       Operator$ = NextTerm$(CurrentString$)
       SELECT CASE Operator$
          CASE "+"
             RPStack! = RPStack! + NewTerm!
          CASE "-"
             RPStack! = RPStack! - NewTerm!
          CASE "*"
             RPStack! = RPStack! * NewTerm!
          CASE "/"
             RPStack! = RPStack! / NewTerm!
          CASE ELSE
             PRINT "Operatore non corretto:"; Operator$
             END
       END SELECT
    LOOP WHILE CurrentString$ <> ""
 END IF
 PRINT "Risultato:", RPStack!
```

Ora è necessario immettere la funzione di analisi *NextTerm$()* il cui scopo è quello di accettare una stringa, ricercarne il primo termine (eventualmente collegata a spazi di riempimento), restituire il termine e troncare la stringa di immissione. Come primo passo si dovrà ricercare la lunghezza del primo termine a sinistra della stringa in modo da poterla troncare.

```
DECLARE FUNCTION NextTerm$ (StringToParse$)

LINE INPUT "Espressione da calcolare:"; CurrentString$
IF LEN(CurrentString$) >= 5 THEN     ' minimo 5 caratteri
   RPStack! = VAL(NextTerm$(CurrentString$))' primo termine
   DO
      NewTerm! = VAL(NextTerm$(CurrentString$))
      Operator$ = NextTerm$(CurrentString$)
      SELECT CASE Operator$
         CASE "+"
            RPStack! = RPStack! + NewTerm!
         CASE "-"
            RPStack! = RPStack! - NewTerm!
         CASE "*"
            RPStack! = RPStack! * NewTerm!
         CASE "/"
            RPStack! = RPStack! / NewTerm!
         CASE ELSE
            PRINT "Operatore non corretto:"; Operator$
            END
      END SELECT
   LOOP WHILE CurrentString$ <> ""
```

```
END IF
PRINT "Risultato:", RPStack!
FUNCTION NextTerm$ (StringToParse$)
     TermLength% = INSTR(StringToParse$, "") - 1 'Trovato spazio?
     :
     :
END FUNCTION
```

Si è così ricercato l'eventuale primo spazio con la funzione *INSTR( )* del BASIC. Per esempio, nella stringa *3.1 2.1 – 10 * 5 +*, *TermLength%* verrà impostata come segue:

TermLength% = 3

3.1 2.1 – 10 * 5 +

Bisogna ora sapere dove tagliare il primo termine e si dovrà anche conoscere la lunghezza rimanente della stringa (in modo da poterla troncare).

```
DECLARE FUNCTION NextTerm$ (StringToParse$)
LINE INPUT "Espressione da calcolare:"; CurrentString$
IF LEN(CurrentString$) >= 5 THEN      ' minimo 5 caratteri
   RPStack! = VAL(NextTerm$(CurrentString$)) ' primo termine
   DO
      NewTerm! = VAL(NextTerm$(CurrentString$))
      Operator$ = NextTerm$(CurrentString$)
      SELECT CASE Operator$
         CASE "+"
            RPStack! = RPStack! + NewTerm!
         CASE "-"
            RPStack! = RPStack! - NewTerm!
         CASE "*"
            RPStack! = RPStack! * NewTerm!
         CASE "/"
            RPStack! = RPStack! / NewTerm!
         CASE ELSE
            PRINT "Operatore non corretto:"; Operator$
            END
      END SELECT
   LOOP WHILE CurrentString$ <> ""
END IF
PRINT "Risultato:", RPStack!
FUNCTION NextTerm$ (StringToParse$)
     TermLength% = INSTR(StringToParse$, "") - 1 ' Trovato spazio?
     NewStringLen% = LEN(StringToParse$)- TermLength% - 1
     :
     :
END FUNCTION
```

Nella stringa *3.1 2.1 – 10 * 5 +* ecco come dovrebbe essere lunga la nuova stringa:

TermLength% = 3

3.1 2.1 – 10 * 5 +

NewStringLen% = 14

Tuttavia, si dovrebbe anche essere in grado di sapere se nella stringa, una volta raggiunta la fine della stessa, non sono stati immessi spazi. In questo caso sarà necessario impostare la lunghezza del primo termine (TermLenght%) alla lunghezza della stringa e impostare la nuova lunghezza della stringa (NewStrngLen%) dopo averla troncata a zero.

```
DECLARE FUNCTION NextTerm$ (StringToParse$)

LINE INPUT "Espressione da calcolare:"; CurrentString$
IF LEN(CurrentString$) >= 5 THEN      ' minimo 5 caratteri
   RPStack! = VAL(NextTerm$(CurrentString$)) ' primo termine
   DO
      NewTerm! = VAL(NextTerm$(CurrentString$))
      Operator$ = NextTerm$(CurrentString$)
      SELECT CASE Operator$
         CASE "+"
            RPStack! = RPStack! + NewTerm!
         CASE "-"
            RPStack! = RPStack! - NewTerm!
         CASE "*"
            RPStack! = RPStack! * NewTerm!
         CASE "/"
            RPStack! = RPStack! / NewTerm!
         CASE ELSE
            PRINT "Operatore non corretto:"; Operator$
            END
      END SELECT
   LOOP WHILE CurrentString$ <> ""
END IF
PRINT "Risultato:", RPStack!

FUNCTION NextTerm$ (StringToParse$)
   TermLength% = INSTR(StringToParse$, "") - 1 ' Trovato spazio?
   NewStringLen% = LEN(StringToParse$)- TermLength% - 1
   IF TermLength% = -1 THEN      ' Raggiunta la fine della stringa
      TermLength% = LEN(StringToParse$)
      NewStringLen% = 0
   END IF
   :
END FUNCTION
```

Si è ora pronti a tagliare il primo termine della stringa di immissione (utilizzando la funzione *LEFT$()*) e a troncare la stringa stessa (utilizzando la funzione *RIGHT$()*), come mostrato nel listato 1.6.

```
DECLARE FUNCTION NextTerm$ (StringToParse$)

LINE INPUT "Espressione da calcolare: "; CurrentString$

IF LEN(CurrentString$) >= 5 THEN        ' Minimo 5 caratteri

      RPStack! = VAL(NextTerm$(CurrentString$)) ' primo termine

        DO
           NewTerm! = VAL(NextTerm$(CurrentString$))
           Operator$ = NextTerm$(CurrentString$)
           SELECT CASE Operator$
                   CASE "+"
                         RPStack! = RPStack! + NewTerm!
                   CASE "-"
                         RPStack! = RPStack! - NewTerm!
                   CASE "*"
                         RPStack! = RPStack! * NewTerm!
                   CASE "/"
                         RPStack! = RPStack! / NewTerm!
                   CASE ELSE
                         PRINT "Operatore scorretto: "; Operator$
                   END
           END SELECT
   LOOP WHILE CurrentString$ <> ""
END IF

PRINT "Risultato: ", RPStack!

FUNCTION NextTerm$ (StringToParse$)
   TermLength% = INSTR(StringToParse$, "") - 1 ' Trovato spazio?
   NewStringLen% = LEN(StringToParse$)- TermLength% - 1
   IF TermLength% = -1 THEN    ' Raggiunta la fine
                               ' della stringa
           TermLength% = LEN(StringToParse$)
           NewStringLen% = 0
   END IF
   NextTerm$ = LEFT$(StringToParse$, TermLength%)
                                     ' Termine successivo
   StringToParse$ = RIGHT$(StringToParse$, NewStringLen%)
END FUNCTION
```

Listato 1.6 *Funzione di analisi di un'immissione*

Con questo si è conclusa l'operazione di taglio del primo termine e di troncamento della stringa. Il termine che è stato isolato viene trasferito nuovamente al programma chiamante e verrà avviato su di esso un *loop* che eseguirà le richieste di moltiplicazione o di somma fino a quando saranno stati utilizzati tutti i termini. A questo punto il risultato verrà emesso a video e terminerà l'operazione di calcolo.

CONCLUSIONE

Si conclude qui il capitolo relativo all'immissione da tastiera. Si sono analizzate le funzioni e i comandi del BASIC, si è visto come integrarle con semplici funzioni personalizzate e come generare una routine di immissione del tipo di quelle che verranno utilizzate in applicazioni reali. Si è visto, inoltre, come eseguire analisi delle immissioni da tastiera. Il capitolo 2 prenderà in esame le procedure di emissione servendosi, tra l'altro, di finestre personalizzate.

CAPITOLO 2

FINESTRE

In questo capitolo verrà presa in considerazione una serie di particolari funzioni e sottoprogrammi che consentiranno la generazione e l'uso di finestre da applicare alle proprie applicazioni. Con poche e semplici chiamate a routine sarà possibile creare a video una finestra contenente i colori specificati, stamparla e nasconderla nuovamente consentendo così di attribuire alle applicazioni un particolare tocco di professionalità. Nella parte finale del capitolo si vedrà anche che il BASIC PDS è in grado di semplificare le procedure di creazione di finestre personalizzate.

PROGETTAZIONE DI UN SISTEMA PERSONALIZZATO DI FINESTRE

Si vedrà ora come progettare e realizzare un sistema basato su finestre multiple. In effetti è molto raro che un'applicazione basata su finestre ne utilizzi una sola e questo significa che sarà necessario creare una procedura che faccia riferimento a una particolare finestra. In linea generale, non viene richiesta la creazione di routine particolari per la generazione di ciascuna finestra (come per esempio, *CALL InitWindow( )*, *CALL InitWindow2( )*, *CALL ShowWindow1( )*, ecc.). Il sistema migliore per creare un sistema a finestre (come quello, per esempio, utilizzato dalle applicazioni di tipo professionale o da *Presentation Manager* di OS/2) è quello di utilizzare gli *identificatori delle finestre*, ossia assegnando un numero a ogni finestra.

Se si intende progettare un sistema personalizzato a finestre da utilizzare in una particolare applicazione composta, per esempio, da una serie di otto finestre, l'identificatore delle finestre si baserà sul numero di indice delle finestre (figura 2.1).

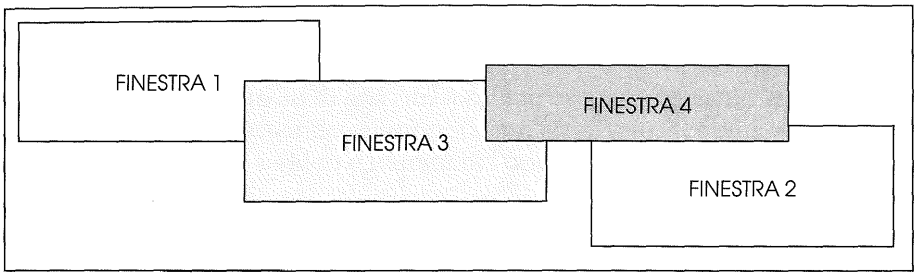

Figura 2.1

Questo concetto renderà più semplice il processo di sviluppo presentato in questo capitolo. Per esempio, per fare apparire a video una finestra contenente il messaggio "BENVENUTI!" e in seguito farla scomparire sarà necessario utilizzare una sequenza procedurale come quella di seguito indicata.

1. Creare una routine di gestione delle finestre alla quale verrà assegnato il nome di *WindowGetHandle%( )*. Questa funzione consentirà di progettare e inizializzare la finestra.

2. Richiamare un sottoprogramma al quale verrà assegnato il nome *WindowShow( )* che visualizzi la finestra. L'unico argomento che il sottoprogramma richiederà è il numero che identifica la finestra.

3. Richiamare una nuova funzione (*WindowPrint%( )*) alla quale viene passato il numero della finestra e il testo in essa contenuto: "BENVENUTI!".

4. Infine, verrà richiamata la funzione *WindowHide( )* alla quale verrà passato il numero identificativo della finestra per farla scomparire dal video.

Ora che è stata impostata la sequenza generale delle operazioni si potrà iniziare a scrivere il programma. Il punto da cui iniziare è la funzione di inizializzazione della finestra *WindowGetHandle%( )*.

INIZIALIZZAZIONE DEL SISTEMA PERSONALIZZATO A FINESTRE

Lo scopo di *WindowGetHandle%( )* è quello di inizializzare una finestra; questa è sempre la prima operazione da compiere. Quando viene progettata e impostata una finestra (dimensione, posizione nello schermo, ecc.) dovrà essere utilizzata la funzione *WindowGetHandle%( )*. Questa funzione dovrà restituire il numero identifica-

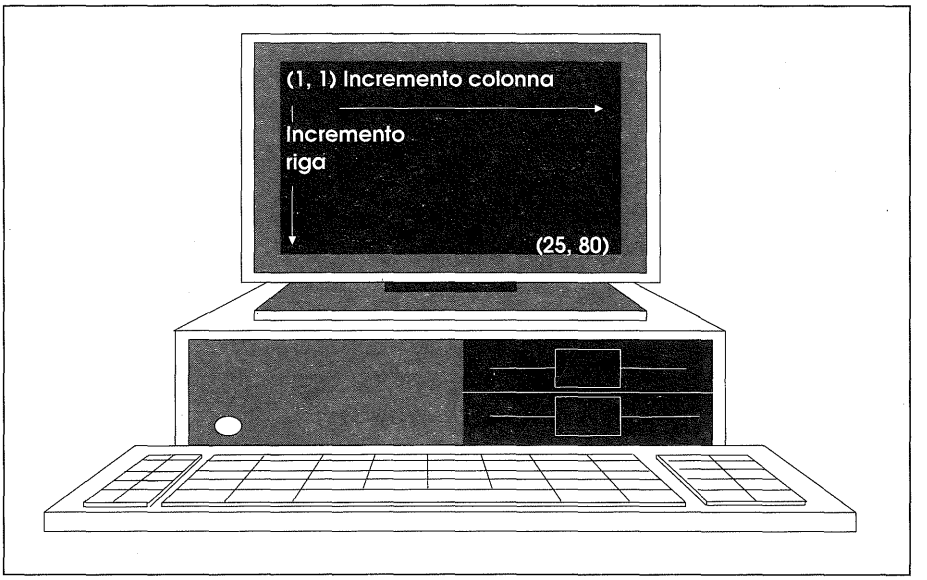

Figura 2.2: *Coordinate dell'angolo superiore sinistro e inferiore destro del video.*

tivo della finestra, che dovrà essere un valore intero contenuto tra 1 e 8, valore che sarà utilizzato in seguito per i riferimenti a questa finestra; in questo modo il sistema sarà in grado di riconoscere a quale delle otto finestre si fa riferimento.

Si noti che l'inizializzazione di una finestra non la farà apparire automaticamente a video poiché la tecnica standard della gestione delle finestre permette di operare sulle finestre senza che queste vengano materialmente visualizzate, oppure di spostarle dove si desidera consentendo così all'utente di avviare azioni in *background*. Per visualizzare sullo schermo una finestra dopo averla inizializzata, è necessario richiamare *WindowShow(Handle%)* dove *Handle%* è il numero identificativo della finestra. Per nascondere nuovamente la finestra, è, invece, necessario richiamare la funzione *WindowHide(Handle%)*.

Per inizializzare una finestra sarà necessario definirne la posizione sullo schermo trasferendo a *WindowGetHandle%( )* le coordinate video dell'angolo superiore sinistro della finestra. In BASIC, l'indicazione *(1, 1)* corrisponde all'angolo superiore sinistro dello schermo e *(25, 80)* corrisponde all'angolo inferiore destro (figura 2.2).

Inoltre, sarà necessario definire la dimensione della finestra trasferendo il numero delle colonne e delle righe comporranno l'interno della finestra. Infine dovrebbe venire assegnato un colore trasferendo un attributo video.

Più avanti verrà presentata una trattazione completa degli attributi video e del loro uso. In altre parole, l'uso che si dovrebbe fare della funzione *WindowGetHandle$( )* è quello di seguito riportato:

```
MyHandle% = WindowGetHandle% (TopR%, TopC%, NRow%, NCol%, Attr%)
```

del quale vengono qui riportati i significati di ciascun parametro.

Nome parametro	*Tipo*	*Descrizione*
TopR%	INTEGER	Riga corrispondente all'angolo superiore sinistro della finestra (*1 = prima riga dello schermo*).
TopC%	INTEGER	Colonna corrispondente all'angolo superiore sinistro della finestra (*1 = prima colonna dello schermo*).
NRow%	INTEGER	Numero di righe corrispondenti all'altezza della finestra.
NCol%	INTEGER	Numero di colonne corrispondenti alla larghezza della finestra.
Attr%	INTEGER	Attributi video che verranno associati alla visualizzazione della finestra.

Per poter utilizzare questi valori, *WindowGetHandle%( )* dovrà assegnare una serie di variabili interne della finestra con i valori che verranno immessi dall'utente. Tutti gli altri sottoprogrammi nel sistema a finestre potranno interfacciarsi tramite *COMMON* che consente la lettura di queste variabili. Se non si ha ben chiaro cosa sta al di sotto dell'istruzione *COMMON* si tenga presente che questa istruzione consente ai vari moduli del programma di poter comunicare fra di loro. Per esempio, in presenza di tre variabili, *A%*, *B%* e *C%*, queste potranno essere utilizzate in tutti i moduli immettendo all'inizio di ciascuno di essi una riga come quella riportata in figura 2.3.

```
COMMON SHARED /MyCommon/ A%, B%, C%
.
.
.
```

Figura 2.3

L'inserimento della parola chiave *SHARED* metterà a disposizione tutte le variabili indicate per tutti i sottoprogrammi e funzioni del modulo corrente. Tuttavia, si tenga presente che *A%*, *B%* e *C%* potranno essere utilizzate in una qualsiasi parte del programma semplicemente immettendone il loro nome. Ogni volta che verrà utilizzata una di queste variabili, si accederà a una locazione di memoria e, perciò, se *A%* verrà modificata da *3* a *5*, il valore *5* sarà quello assunto all'interno di tutte le parti del programma. Con l'uso delle istruzioni *COMMON* tutti i sottoprogrammi che

generano e gestiscono finestre potranno operare indipendentemente l'uno dall'altro, il che significa che sarà sufficiente collegare alla finestra solo le funzioni che si desiderano utilizzare e non l'intero sistema. In questo modo si potranno sviluppare le finestre all'interno di sottoprogrammi con poche righe di programma.

> ***Consiglio:*** L'operazione di suddivisione dei programmi in unità più brevi facilmente gestibili viene definita *programmazione modulare.* La programmazione modulare facilita le operazioni di sviluppo e controllo dei vari frammenti di codice. L'istruzione *COMMON* è, in sostanza, la spina dorsale della programmazione modulare e, proprio per questa ragione, è eccezionalmente versatile.

COMPILAZIONE DELL'ARRAY DELLE FINESTRE

Si inizierà ora con l'impostazione dell'array che dovrà contenere i dati della finestra. Nell'esempio che viene qui sviluppato, sarà necessario un array (al quale verrà assegnato il nome *Attribute( )*) che dovrà contenere gli attributi video di ciascuna finestra. Se si desiderano creare otto finestre, l'indice di *Attribute( )* dovrà prevedere un intervallo che va da 1 a 8. Poiché, probabilmente, in un secondo momento si vorranno creare più di otto finestre è consigliabile non utilizzare il valore costante "*8*" ma assegnarlo a una variabile (il cui nome potrebbe essere *MaxWindows*) che potrà essere modificata in qualsiasi momento.

```
DECLARE FUNCTION WindowGetHandle% (TopR%, TopC%, NRow%, NCol%,⌉
                                   Attr%)

CONST MaxWindows = 8
:
:
```

Più avanti, se si vorrà aumentare o diminuire il numero delle finestre a disposizione, questo sarà l'unico valore da modificare (senza alcun limite apparente). Si tenga presente, tuttavia, che un numero elevato di finestre corrisponde a un utilizzo maggiore della memoria.

Il passo successivo è quello di memorizzare i dati che sono stati trasferiti. Dovrà essere creato un array per ogni variabile della finestra; la dimensione dell'array corrisponderà al numero massimo di finestre previste (*MaxWindows*) e l'indice dell'array coinciderà con il numeo identificativo della finestra: se, per esempio si richiedesse la visualizzazione della finestra 3, gli attributi di tale finestra saranno contenuti in *Attribute(3).*

Un importante elemento di ogni finestra è rappresentato dal numero di colonne e di righe che la compongono e questi valori potranno essere definiti come riportato di seguito.

```
DECLARE FUNCTION WindowGetHandle% (TopR%, TopC%, NRow%, NCol%,⌉
                                   Attr%)

    CONST MaxWindows = 8
    DIM Rows(1 TO MaxWindows) AS INTEGER
    DIM Cols(1 TO MaxWindows) AS INTEGER
    :
    :
```

Successivamente, si potranno immettere le coordinate di ogni finestra:

```
FUNCTION WindowGetHandle% (TopR%, TopC%, NRow%, NCol%,⌉
                           Attr%)

    CONST MaxWindows = 8
    DIM Rows(1 TO MaxWindows) AS INTEGER
    DIM Cols(1 TO MaxWindows) AS INTEGER
    DIM Toprow(1 TO MaxWindows) AS INTEGER
    DIM TopCol(1 TO MaxWindows) AS INTEGER
    DIM BotRow(1 TO MaxWindows) AS INTEGER
    DIM BotCol(1 TO MaxWindows) AS INTEGER
    :
    :
```

In questo caso l'angolo superiore sinistro della finestra è (TopRow, TopCol) e quello inferiore destro è (BotRow, BotCol). In base a ciò, gli attributi della finestra potranno essere impostati come segue:

```
DECLARE FUNCTION WindowGetHandle% (TopR%, TopC%, NRow%, NCol%,⌉
                                   Attr%)

    CONST MaxWindows = 8
    DIM Rows(1 TO MaxWindows) AS INTEGER
    DIM Cols(1 TO MaxWindows) AS INTEGER
    DIM Toprow(1 TO MaxWindows) AS INTEGER
    DIM TopCol(1 TO MaxWindows) AS INTEGER
    DIM BotRow(1 TO MaxWindows) AS INTEGER
    DIM BotCol(1 TO MaxWindows) AS INTEGER
    DIM Attribute(1 TO MaxWindows) AS INTEGER
    :
    :
```

Ora è il momento di immettere il testo che sarà contenuto nella finestra definendo ogni riga della finestra con una stringa. Si imposti un array bidimensionale a cui assegnare il nome *Text(MaxWindows, 25) AS STRING* in modo che esso sia in grado di contenere le stringhe che verranno definite. Si supponga che la finestra 3 sia quella mostrata in figura 2.4.

```
È arrivato il
momento per vedere
i risultati
```

Figura 2.4

Si assegni quindi a *Text(3, 1)* la stringa "È arrivato il", a *Text(3, 2)* la stringa "momento per vedere" ecc. Si è così costruito un array (si tenga presente che in una finestra potranno essere contenute al massimo 25 righe).

```
DECLARE FUNCTION WindowGetHandle% (TopR%, TopC%, NRow%,⌉
                                   NCol%, Attr%)

   CONST MaxWindows = 8
   DIM Rows(1 TO MaxWindows) AS INTEGER
   DIM Cols(1 TO MaxWindows) AS INTEGER
   DIM Toprow(1 TO MaxWindows) AS INTEGER
   DIM TopCol(1 TO MaxWindows) AS INTEGER
   DIM BotRow(1 TO MaxWindows) AS INTEGER
   DIM BotCol(1 TO MaxWindows) AS INTEGER
   DIM Attribute(1 TO MaxWindows) AS INTEGER
   DIM Text(MaxWindows, 25) AS STRING
   :
   :
```

Si noti che quando la finestra verrà nuovamente nascosta sarà necessario ripristinarvi il testo presente a video. Questo significa che sarà necessario salvare il testo che è stato sovrascritto quando la finestra è stata visualizzata. Il testo potrà essere memorizzato in un array a cui assegnare il nome *OldText( )*.
Inoltre, a ogni posizione video dovrà essere assegnato un particolare attributo che determini il colore del carattere; ciò significa che anche gli attributi video dovranno essere memorizzati con il testo. Poiché si prevede che lo schermo sia in grado di rappresentare vari colori (che potrebbero anche venire sovrapposti allo sfondo colorato delle finestre) sarà necessario memorizzare, insieme al testo, anche gli attributi di *ciascuna* posizione video che viene sovrascritta. Poiché un attributo video corrisponde a un byte, anche questo elemento potrà essere memorizzato in una stringa.

```
DECLARE FUNCTION WindowGetHandle% (TopR%, TopC%, NRow%, NCol%,⌉
                                   Attr%)

   CONST MaxWindows = 8
   DIM Rows(1 TO MaxWindows) AS INTEGER
   DIM Cols(1 TO MaxWindows) AS INTEGER
   DIM Toprow(1 TO MaxWindows) AS INTEGER
   DIM TopCol(1 TO MaxWindows) AS INTEGER
   DIM BotRow(1 TO MaxWindows) AS INTEGER
```

```
DIM BotCol(1 TO MaxWindows) AS INTEGER
DIM Attribute(1 TO MaxWindows) AS INTEGER
DIM Text(MaxWindows, 25) AS STRING
DIM OldText(MaxWindows, 25) AS STRING
DIM OldAttrb(MaxWindows, 25) AS STRING
:
:
```

La routine è così quasi terminata. Sono state memorizzate le specifiche della finestra, compresi dimensione e colore. Si è creato uno spazio per immettere un testo all'interno della finestra e uno spazio di backup del testo e degli attributi video che verranno sovrascritti.
L'ultima cosa da aggiungere è una variabile interna che indichi se la finestra dovrà essere visualizzata o meno (se non verrà immesso questo valore, verranno visualizzate contemporaneamente tutte le finestre).
Sarà quindi necessario immettere un *flag* (*OnFlag( )*) che indichi se la finestra dovrà essere visualizzata. Nel caso, per esempio, in cui *OnFlag(5)* sia pari a 1, verrà visualizzata la finestra 5, nel caso invece in cui sia pari a 0, la finestra verrà nascosta.

```
DECLARE FUNCTION WindowGetHandle% (TopR%, TopC%, NRow%, NCol%,
                                   Attr%)

CONST MaxWindows = 8
DIM Rows(1 TO MaxWindows) AS INTEGER
DIM Cols(1 TO MaxWindows) AS INTEGER
DIM Toprow(1 TO MaxWindows) AS INTEGER
DIM TopCol(1 TO MaxWindows) AS INTEGER
DIM BotRow(1 TO MaxWindows) AS INTEGER
DIM BotCol(1 TO MaxWindows) AS INTEGER
DIM Attribute(1 TO MaxWindows) AS INTEGER
DIM Text(1 TO MaxWindows) AS STRING
DIM OldText(1 TO MaxWindows) AS STRING
DIM OldAttrb(1 TO MaxWindows) AS STRING
DIM OnFlag(1 TO MaxWindows) AS INTEGER
:
:
```

E con questo si è terminato di impostare il sistema a finestre. Quello che resta da fare è impostare le istruzioni *COMMON* per rendere comuni questi array agli altri sottoprogrammi e in seguito riempire gli array con i valori appropriati.
Il listato 2.1 mostra la funzione completa.

```
DECLARE FUNCTION WindowGetHandle% (TopR%, TopC%, NRow%, NCol%,⌉
                                   Attr%)

    CONST MaxWindows = 8
    DIM Rows(1 TO MaxWindows) AS INTEGER
    DIM Cols(1 TO MaxWindows) AS INTEGER
    DIM Toprow(1 TO MaxWindows) AS INTEGER
    DIM TopCol(1 TO MaxWindows) AS INTEGER
    DIM BotRow(1 TO MaxWindows) AS INTEGER
    DIM BotCol(1 TO MaxWindows) AS INTEGER
    DIM Attribute(1 TO MaxWindows) AS INTEGER
    DIM OnFlag(1 TO MaxWindows) AS INTEGER
    DIM Text(MaxWindows, 25) AS STRING
    DIM OldText(MaxWindows, 25) AS STRING
    DIM OldAttrb(MaxWindows, 25) AS STRING

    COMMON SHARED /WindowA/ Rows() AS INTEGER, Cols() AS INTEGER,⌉
       Toprow() AS INTEGER, TopCol() AS INTEGER,⌉
       BotRow() AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /WindowB/ Attribute() AS INTEGER,
       OnFlag() AS INTEGER, Text() AS STRING, OldText() AS STRING,⌉
       OldAttrb() AS STRING

FUNCTION WindowGetHandle% (TopR%, TopC%, NRow%, NCol%, Attr%)
    WindowGetHandle% = 0

    FOR i = 1 TO MaxWindows
       IF Rows(i) = 0 THEN
          WindowGetHandle% = i
          Toprow(i) = TopR%
          TopCol(i) = TopC%
          BotRow(i) = TopR% + NRow%
          BotCol(i) = TopC% + NCol%
          Rows(i) = NRow%
          Cols(i) = NCol%
          Attribute(i) = Attr%
          OnFlag(i) = 0
          FOR j = 1 TO Rows(i)
             Text(i, j) = SPACE$(Cols(i))
          NEXT j
       EXIT FOR
       END IF
    NEXT i

END FUNCTION
```

Listato 2.1 *Funzione WindowGetHandle*

Si noti, ancora una volta, che il numero identificativo della finestra corrisponde all'indice nell'array della finestra stessa. Quando verrà richiamata *WindowGetHandle%( )* sarà necessario reperire il primo indice di array disponibile da assegnare alla nuova finestra e restituirlo come numero identificativo. Questa è un'operazione che potrà essere eseguita controllando *Rows(i)*, il primo degli array finestra.

```
FOR i = 1 TO MaxWindows
   IF Rows(i) = 0 THEN
      WindowGetHandle% = i
      Toprow(i) = TopR%
      TopCol(i) = TopC%
      BotRow(i) = TopR% + NRow%
      BotCol(i) = TopC% + NCol%
      Rows(i) = NRow%
      Cols(i) = NCol%
      Attribute(i) = Attr%
      OnFlag(i) = 0
      FOR j = 1 TO Rows(i)
         Text(i, j) = SPACE$(Cols(i))
      NEXT j
   EXIT FOR
   END IF
NEXT i
```

Se, per esempio, si scopre che *Rows(4)* è pari a 0, ciò significa che non esiste alcuna finestra 4 (in quanto non è possibile avere una finestra con un valore di righe pari a zero), per cui il valore verrà restituito come identificativo di finestra necessario per riempire gli array con diversi valori (per esempio: *Rows(4) Cols(4)* ecc.). A questo punto si è terminata la scrittura della funzione *WindowGetHandle%( )* e sono stati memorizzati tutti i dati necessari alla finestra completando così il primo passo verso lo sviluppo di un sistema basato su finestre. Si proverà ora ad attivare la funzione.

ESEMPIO DI INIZIALIZZAZIONE DI UNA FINESTRA

Per preparare l'esempio che utilizza *WindowGetHandle%( )* è necessario immettere correttamente i valori che verranno via via richiesti secondo quanto di seguito riportato:

Variabile	*Tipo*	*Descrizione*
TopR%	INTEGER	Riga corrispondente all'angolo superiore sinistro della finestra (*1 = prima riga dello schermo*).
TopC%	INTEGER	Colonna corrispondente all'angolo superiore sinistro della finestra (*1 = prima colonna dello schermo*).

NRow%	INTEGER	Numero di righe corrispondenti all'altezza della finestra.
NCol%	INTEGER	Numero di colonne corrispondenti alla larghezza della finestra.
Attr%	INTEGER	Attributi video che verranno associati alla visualizzazione della finestra.

In particolare, sarà necessario immettere un *attributo video* che verrà trasferito sotto forma di *INTEGER* anche se verranno utilizzati solo gli 8 bit di ordine inferiore. In effetti questo byte definisce il colore di sfondo della finestra e del testo in essa contenuto. Per esempio, si potranno selezionare caratteri verdi su sfondo blu, oppure caratteri gialli su sfondo rosso.

***Nota**: L'impostazione del bit a 1 genera colori di tipo particolare.*

La figura 2.5 rappresenta graficamente la composizione di un byte dell'attributo video.

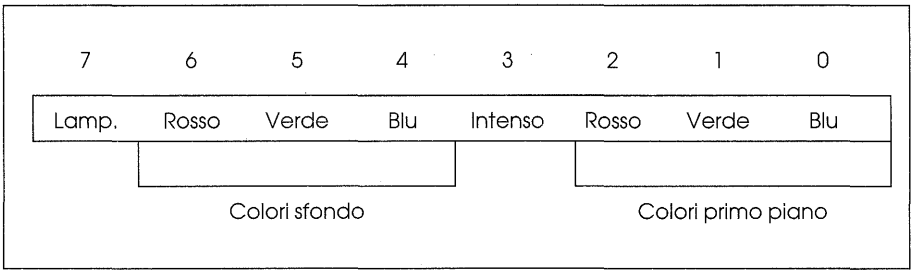

Figura 2.5

L'impostazione del byte di attributo video consente di selezionare il rosso, il verde e il blu sia per i colori di sfondo (il colore di sfondo della finestra) sia per quelli di primo piano (il colore dei caratteri). Inoltre sarà possibile impostare il bit 3 in modo che attivi l'alta intensità e il bit 7 per attivare il lampeggiamento del carattere. Un primo piano rosso su sfondo verde dovrebbe corrispondere a un byte di attributo video 00100100 (notazione binaria) oppure a &H24 (figura 2.6).
I colori potranno essere miscelati effettuando la somma dei rispettivi bit. Di seguito vengono riportati i valori dei bit che sommati formano il byte di attributo video.

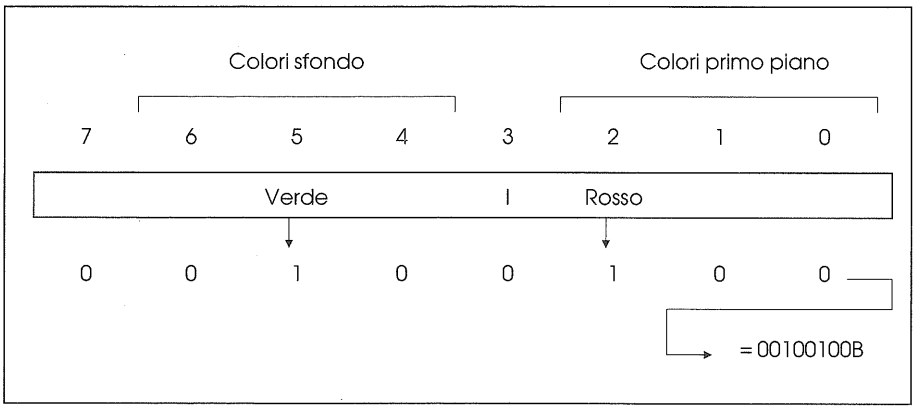

Figura 2.6

Valore bit	*Colore risultante*
1	Primo piano blu
2	Primo piano verde
4	Primo piano rosso
8	Alta intensità
16	Sfondo blu
32	Sfondo verde
64	Sfondo rosso
128	Lampeggiamento

Per esempio, per impostare una normale visualizzazione di bianco su sfondo nero, sarà necessario attivare tutti i colori di primo piano (corrispondenti alle lettere) utilizzando *1* + *2* + *4* = *7*. Tutti i colori di sfondo dovranno, invece, essere disattivati, ovvero impostati a 0. Il valore di 7 è il normale valore di attributo video iniziale e se si desidera convertirlo in alta intensità, sarà necessario aggiungervi 8 per ottenere 15, che corrisponde a &HF. Se, al contrario, si desidera un lampeggiamento su sfondo bianco, sarà necessario impostare attivare tutti i colori di sfondo e aggiungere 128 per generare il lampeggiamento: 16 + 32 + 64 + 128 = 240 che corrisponde a &HF0. Su uno schermo monocromatico non sarà possibile utilizzare i colori singoli ma dovrà essere attivato lo schermo normale (7), l'alta intensità (&HF), il lampeggiamento normale (&H87), il lampeggiamento su video in negativo (&HF0) e la sottolineatura, che non viene gestita dagli schermi grafici. Per attivare la sottolineatura utilizzare il blu come colore di primo piano (assegnandogli un attributo 1), Utilizzando un attributo 9, sarà anche possibile generare una sottolineatura ad alta intensità.

***Consiglio**:* L'interrupt BIOS &H10, servizio 9 consente di selezionare gli attributi video per ciascun carattere man mano che questi verranno visualizzati. Questo interrupt è in grado di effettuare l'operazione con un maggior controllo rispetto a quello disponibile nella combinazione COLOR/PRINT del BASIC, che, da parte sua, permette di assegnare un solo colore all'intera stringa che verrà visualizzata.

Dopo aver passato questa informazione, la funzione *WindowsGetHandle%( )* restituirà l'identificativo della finestra (*Handle%*) che è rappresentato da un valore intero compreso tra 1 e 8. Per visualizzare sullo schermo questa finestra sarà necessario richiamare la funzione *WindowShow(Handle%)*; per rimuoverla, invece, si deve richiamare la funzione *WindowHide(Handle%)*. In altre parole, d'ora in poi, tutto ciò che dovrà essere trasferito ai sottoprogrammi sono gli identificatori di finestra. Non sarà più necessario definire dimensioni, posizioni o attributi video.
Nell'esempio riportato di seguito vengono impostate due finestre: la prima con l'angolo superiore sinistro a *(1, 1)*, con dimensione di cinque colonne e cinque righe e con un attributo video &H21; la seconda con l'angolo superiore sinistro impostato a *(10, 10)*, anch'essa di cinque colonne per cinque righe e con attributo video &H61.

```
DECLARE FUNCTION WindowGetHandle% (TopRow%, TopCol%,
                                   NumRow%, NumCol%, Attr%)

Handle% = WindowGetHandle%(1, 1, 5, 5, &H21)

PRINT "L'identificatore della finestra è: ", Handle%

Handle2% = WindowGetHandle%(10, 10, 5, 5, &h61)

PRINT "L'identificatore della finestra è: ", Handle2%
```

L'identificatore della prima finestra sarà *1* e l'identificatore della seconda sarà *2*. Si è così avuta la possibilità di emulare il processo di inizializzazione di un'applicazione a finestre di tipo professionale. Il prossimo passaggio sarà quello di riportare a video la finestra.

VISUALIZZAZIONE DI UNA FINESTRA

Ora che è stata creata la funzione *WindowGetHandle%( )* si potrà operare sulla finestra sia che essa sia nascosta o che sia visualizzata. Verrà creato un sottoprogramma che visualizza la finestra (*WindowShow( )*); per visualizzare la finestra, nella posizione indicata in *WindowGetHandle%( )* si dovrebbe richiamare *WindowShow(Handle%)*, dove *Handle%* corrisponde all'identificatore della finestra.

Innanzitutto sarà necessario salvare il testo che verrà coperto dalla finestra; si ricorderà che lo spazio per questo salvataggio è già stato impostato dall'array *OldText( )*. Sarà inoltre necessario salvare anche gli attributi video di *ogni* posizione che verranno e memorizzati nell'array *OldAttrb( )*. Questa operazione potrà essere eseguita contemporaneamente attivando un *loop* sulle righe e colonne che verranno sovrascritte a video e utilizzando la funzione *SCREEN* del BASIC (in grado di leggere sia gli attributi che il testo) per poterne leggere il contenuto.

```
SUB WindowShow (Handle%)

   REM Salva la precedente sezione video

   FOR i = 1 TO Rows(Handle%)
      FOR j = 1 TO Cols (Handle%)
      :
      :
      NEXT j
   NEXT i
   :
   :
```

Il testo e gli attributi dei dati potranno essere salvati in stringhe temporanee *temp1$* e *temp2$*:

```
SUB WindowShow (Handle%)

   REM Salva la precedente selezione video

   TR = TopRow(Handle%)
   TC = TopCol(Handle%)

      FOR i = 1 TO Rows(Handle%)
         temp1$ = ""
         temp2$ = ""
         FOR j = 1 TO Cols(Handle%)
            temp1$ = temp1$ + CHR$(SCREEN(TR + i - 1, TC + j - 1))
            temp2$ = temp2$ + CHR$(SCREEN(TR + i - 1, TC + j - 1, 1))
      NEXT j
   NEXT i
   :
   :
```

Poi, per ogni riga, sarà necessario memorizzare la stringa temporanea completa nell'array *OldText( )* e *OldAttrb( )*.

```
SUB WindowShow (Handle%)

   REM Salva la precedente selezione video
```

```
TR = TopRow(Handle%)
TC = TopCol(Handle%)

   FOR i = 1 TO Rows(Handle%)
      temp1$ = ""
      temp2$ = ""
      FOR j = 1 TO Cols(Handle%)
         temp1$ = temp1$ + CHR$(SCREEN(TR + i - 1, TC + j - 1))
         temp2$ = temp2$ + CHR$(SCREEN(TR + i - 1, TC + j - 1, 1))
   NEXT j
   OldText(Handle%, i) = temp1$
   OldAttrb(Handle%, i) = temp2$
NEXT i
:
:
```

Quando lo schermo verrà ripristinato verranno letti i precedenti caratteri e attributi prelevati dagli array sopra indicati. È ora necessario visualizzare la finestra sullo schermo. Si potrà utilizzare l'interrupt &H10, servizio 9 che permette di visualizzare sia i caratteri che i loro attributi alla posizione del cursore. Nell'Appendice di questo libro viene riportata l'intera descrizione dell'interrupt &H10, servizio 9.

***Nota**: Non sarà possibile utilizzare l'istruzione PRINT del BASIC, né l'istruzione PRINT USING poiché queste non consentono questo tipo di emissione a video dei caratteri.*

Input		**Output**
ah=	9	Carattere emesso a video alla posizione del cursore.
al=	codice ASCII IBM	
bh=	Numero pagina	
bl →	Modalità testo = Attributo	
	Modalità grafica = Colore	
cx=Conteggio caratteri da scrivere		

Per utilizzare questo servizio è necessario caricare *ah* con 9, *al* con il codice ASCII del carattere (ottenibile da *Text( )*) e *bh* con il *numero pagina*. Lo schermo di output potrà essere suddiviso in varie *pagine*, anche se si utilizzerà sempre la pagina di default (quella visualizzata), ovvero la pagina 0.
Sarà anche necessario caricare *bl* con l'attributo assegnato al carattere (ottenibile da *Attribute( )*) e *cx* con il numero dei caratteri da stampare *(1)*. Per spostare il cursore all'interno dello schermo, si potrà utilizzare l'istruzione *LOCATE*.

```
SUB WindowShow (Handle%)

   REM Salva la precedente selezione video

   TR = TopRow(Handle%)
   TC = TopCol(Handle%)

      FOR i = 1 TO Rows(Handle%)
         temp1$ = ""
         temp2$ = ""
         FOR j = 1 TO Cols(Handle%)
            temp1$ = temp1$ + CHR$(SCREEN(TR + i - 1, TC + j - 1))
            temp2$ = temp2$ + CHR$(SCREEN(TR + i - 1, TC + j - 1, 1))
      NEXT j
      OldText(Handle%, i) = temp1$
      OldAttrb(Handle%, i) = temp2$
   NEXT i

   REM Stampa del nuovo testo

   LOCATE TR, TC

   FOR i = 1 TO Rows(Handle%)
      FOR j = 1 TO Cols(Handle%)
      :
      NEXT j
   NEXT i
```

Poi è necessario spostare il cursore nella sua corretta posizione (si noti che si è iniziato con il salvataggio della posizione attuale del cursore in *CurRow%*e *CurRow%* che consente quindi di richiamare il cursore dopo che l'operazione è terminata.

```
SUB WindowShow (Handle%)

   CurRow% = CSRLIN
   CurCol% = POS(0)

   REM Salva la precedente selezione video

   TR = TopRow(Handle%)
   TC = TopCol(Handle%)

      FOR i = 1 TO Rows(Handle%)
         temp1$ = ""
         temp2$ = ""
         FOR j = 1 TO Cols(Handle%)
            temp1$ = temp1$ + CHR$(SCREEN(TR + i - 1, TC + j - 1))
            temp2$ = temp2$ + CHR$(SCREEN(TR + i - 1, TC + j - 1, 1))
      NEXT j
```

```
      OldText(Handle%, i) = temp1$
      OldAttrb(Handle%, i) = temp2$
   NEXT i

   REM Stampa del nuovo testo

   LOCATE TR, TC    ' Inizia all'angolo superiore sinistro
                    ' della finestra

   FOR i = 1 TO Rows(Handle%)
      FOR j = 1 TO Cols(Handle%)
         :
         [Immette a video il carattere]
         :
         LOCATE TR + i - 1, TC + j     ' Posizionamento sulle
                                       ' coordinate del carattere
                                       ' successivo
      NEXT j
         LOCATE TR + i, TC             ' Salta alla riga successiva
   NEXT i
```

Successivamente si impostino i registri e si utilizzi l'interrupt &H21, servizio 9.

```
SUB WindowShow (Handle%)

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   REM Salva la precedente selezione video

   TR = TopRow(Handle%)
   TC = TopCol(Handle%)

      FOR i = 1 TO Rows(Handle%)
         temp1$ = ""
         temp2$ = ""
         FOR j = 1 TO Cols(Handle%)
            temp1$ = temp1$ + CHR$(SCREEN(TR + i - 1, TC + j - 1))
            temp2$ = temp2$ + CHR$(SCREEN(TR + i - 1, TC + j - 1, 1))
      NEXT j
      OldText(Handle%, i) = temp1$
      OldAttrb(Handle%, i) = temp2$
   NEXT i

   REM Stampa del nuovo testo

   LOCATE TR, TC
```

```
InRegs.cx = 1

FOR i = 1 TO Rows(Handle%)
   FOR j = 1 TO Cols(Handle%)
      InRegs.ax = &H900 + ASC(MID$(Text(Handle%, i), j, 1))
      InRegs.bx = Attribute(Handle%)
      CALL INTERRUPT(&H10, InRegs, OutRegs)
      LOCATE TR + i - 1, TC + j
   NEXT j
      LOCATE TR + i, TC
NEXT i
```

Il passo finale è quello di indicare che la finestra deve essere visualizzata per gli altri sottoprogrammi impostando *OnFlag(Handle%)* a 1 e in seguito reimpostando la posizione del cursore prelevandola da *CurRow%* e *CurCol%*.

> ***Consiglio***: L'impostazione dei *flag* — come per esempio *OnFlag( )* — in una routine SHARED COMMON consente ai moduli di un programma di comunicare l'uno con l'altro *senza* dover trasferire nuovamente parametri tra una chiamata e la successiva.

La finestra ora dovrebbe essere visibile: il listato 2.2 mostra l'intero sottoprogramma *WindowShow( )*.

```
DECLARE SUB WindowShow (Handle%)

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

    CONST MaxWindows = 8
    DIM Rows(1 TO MaxWindows) AS INTEGER
    DIM Cols(1 TO MaxWindows) AS INTEGER
    DIM TopRow(1 TO MaxWindows) AS INTEGER
```

continua

Listato 2.2 *Sottoprogramma WindowShow( ) per visualizzare le finestre.*

```
    DIM TopCol(1 TO MaxWindows) AS INTEGER
    DIM BotRow(1 TO MaxWindows) AS INTEGER
    DIM BotCol(1 TO MaxWindows) AS INTEGER
    DIM Attribute(1 TO MaxWindows) AS INTEGER
    DIM OnFlag(1 TO MaxWindows) AS INTEGER
    DIM Text(MaxWindows, 25) AS STRING
    DIM OldText(MaxWindows, 25) AS STRING
    DIM OldAttrb(MaxWindows, 25) AS STRING
    COMMON SHARED /WindowA/ Rows() AS INTEGER, Cols() AS INTEGER,⌉
         TopRow() AS INTEGER, TopCol() AS INTEGER,⌉
         BotRow() AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /WindowB/ Attribute() AS INTEGER,⌉
         OnFlag() AS INTEGER, Text() AS STRING,⌉
         OldText() AS STRING, OldAttrb() AS STRING

SUB WindowShow (Handle%)
    DIM InRegs AS RegType, OutRegs AS RegType

    CurRow% = CSRLIN
    CurCol% = POS(0)

    REM Salva la precedente sezione video
    TR = TopRow(Handle%)
    TC = TopCol(Handle%)

       FOR i = 1 TO Rows(Handle%)
          temp1$ = ""
          temp2$ = ""
          FOR j = 1 TO Cols(Handle%)
             temp1$ = temp1$ + CHR$(SCREEN(TR + i - 1, TC + j - 1))
             temp2$ = temp2$ + CHR$(SCREEN(TR + i - 1, TC + j⌉
                                          - 1, 1))
       NEXT j
       OldText(Handle%, i) = temp1$
       OldAttrb(Handle%, i) = temp2$
    NEXT i

    REM Stampa del nuovo testo
    LOCATE TR, TC
    InRegs.cx = 1
    FOR i = 1 TO Rows(Handle%)
       FOR j = 1 TO Cols(Handle%)
          InRegs.ax = &H900 + ASC(MID$(Text(Handle%, i), j, 1))
          InRegs.bx = Attribute(Handle%)
          CALL INTERRUPT(&H10, InRegs, OutRegs)
```

continua

Listato 2.2 *Sottoprogramma WindowShow( ) per visualizzare le finestre.*

```
        LOCATE TR + i - 1, TC + j
      NEXT j
        LOCATE TR + i, TC
    NEXT i

    REM Visualizzazione finestra attivata.

    OnFlag(Handle%) = 1

    LOCATE CurRow%, CurCol%

END SUB
```

Listato 2.2 *Sottoprogramma WindowShow( ) per visualizzare le finestre.*

Questo è un sottoprogramma relativamente lungo, anche se, in effetti, è molto facile da utilizzare. Per visualizzare una particolare finestra sarà sufficiente richiamarla tramite il relativo identificatore. La prossima procedura che viene presentata esegue questa operazione.

PROVA DI VISUALIZZAZIONE DI UNA FINESTRA

Viene qui riportata la procedura per visualizzare due finestre a video (nelle quali non è contenuto alcun testo). Prima di tutto le finestre verranno inizializzate con la funzione *WindowGetHandle%( )* e in seguito verrà richiamata *WindowShow( )* con i relativi identificatori.

```
DECLARE FUNCTION WindowGetHandle% (TopRow%, TopCol%, NumRow%,
                 NumCol%, Attr%)
DECLARE SUB WindowShow (Handle%)

Handle1% = WindowGetHandle%(10, 10, 5, 5, &H24)
Handle2% = WindowGetHandle%(20, 20, 3, 13, &H61)

CALL WindowShow(Handle1%)
CALL WindowShow(Handle2%)
```

E per ora è tutto. Entrambe le finestre verranno riportate a video: la prima sarà verde e la seconda marrone (figura 2.7). Fino ad ora il codice ha funzionato correttamente generando la visualizzazione delle finestre; adesso le finestre andranno rimosse.

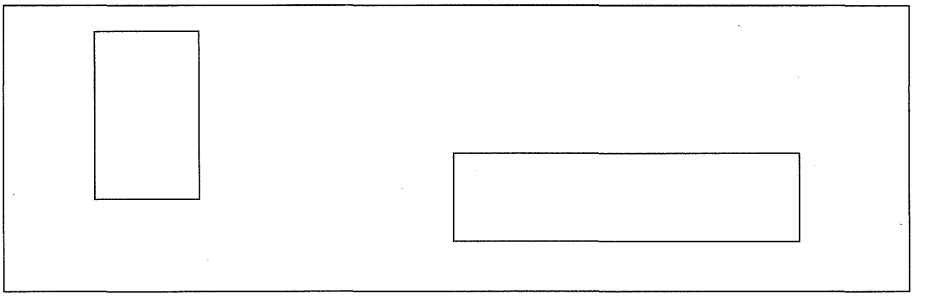

Figura 2.7

RIMOZIONE DI UNA FINESTRA

La funzione *WindowHide( )* è esattamente l'opposto della funzione *WindowShow( )*. La sua funzione è quella di rimuovere le finestre a video e di ripristinare quanto vi era precedentemente sotto di essa. Questa operazione, tuttavia, *non* impedisce di lavorare all'interno della finestra: sarà ancora possibile spostarsi all'interno di essa o generare un'emissione a video, con la sola limitazione che il risultato non sarà visibile. Per rimuovere una finestra sarà sufficiente richiamare *WindowHide( )* associata al relativo identificatore di finestra: per esempio, *CALL WindowHide(Handle%)*. Ora verrà sviluppato questo sottoprogramma: la prima operazione da effettuare è quella di salvare la posizione del cursore (perché sarà necessario riattivarlo alla medesima posizione di partenza quando verrà ripristinato il contenuto del video nascosto dalla finestra).

```
    CurRow% = CSRLIN
    CurCol% = POS(0)

    DIM InRegs AS RegType, OutRegs AS RegType
    :
    :
END SUB
```

Ora sarà necessario generare un *loop* sulle righe presenti nella finestra visualizzata:

```
    CurRow% = CSRLIN
    CurCol% = POS(0)

    DIM InRegs AS RegType, OutRegs AS RegType

    REM Stampa il testo precedente
```

```
LOCATE TR, TC
InRegs.cx = 1

FOR i = 1 TO Rows(Handle%)
   :
   :
NEXT i
```

In seguito si genererà un *loop* interno sulle colonne. Ad ogni posizione verrà ripristinato il carattere precedente e il relativo attributo. Si è già visto come eseguire questa operazione con *WindowPrint( )*: la sola differenza, in questo caso, è data dal fatto che verrà utilizzato un array *OldText( )* al posto dell'array *Text( )* e l'array *OldAttrb( )* al posto di *Attribute( )* per poter ripristinare l'attributo assegnato ad ogni carattere precedentemente a video.

```
CurRow% = CSRLIN
CurCol% = POS(0)

DIM InRegs AS RegType, OutRegs AS RegType

REM Stampa il testo precedente

LOCATE TR, TC
InRegs.cx = 1

FOR i = 1 TO Rows(Handle%)
   FOR j = 1 TO Cols(Handle%)
      InRegs.ax = &H900 + ASC(MID$(OldText(Handle%, i), j, 1))
      InRegs.bx = ASC(MID$(OldAttrb(Handle%, i), j, 1))
      CALL INTERRUPT(&H10, InRegs, OutRegs)
      LOCATE TR + i - 1, TC  + j
   NEXT j
      LOCATE TR + i, TC
NEXT i
   :
   :
```

Quando si sarà terminata l'operatività dei due *loop* la finestra sarà scomparsa dal video. Per terminare sarà necessario indicare che la finestra è disattivata impostando *OnFlag(Handle%)* a 0 e ripristinando la posizione del cursore precedente all'apertura della finestra servendosi dell'istruzione *LOCATE CurRow%, CurCol%*.
A questo punto la finestra è definitivamente scomparsa e il cursore viene riportato alla sua posizione originaria. Il listato 2.3 presenta la funzione completa.

```
DECLARE SUB WindowHide (Handle%)

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,⌉
                       OutRegs AS RegType)
    CONST MaxWindows = 8
    DIM Rows(1 TO MaxWindows) AS INTEGER
    DIM Cols(1 TO MaxWindows) AS INTEGER
    DIM TopRow(1 TO MaxWindows) AS INTEGER
    DIM TopCol(1 TO MaxWindows) AS INTEGER
    DIM BotRow(1 TO MaxWindows) AS INTEGER
    DIM BotCol(1 TO MaxWindows) AS INTEGER
    DIM Attribute(1 TO MAXWINDOWS) AS INTEGER
    DIM OnFlag(1 TO MaxWindows) AS INTEGER
    DIM Text(MaxWindows, 25) AS STRING
    DIM OldText(MaxWindows, 25) AS STRING
    DIM OldAttrb(MaxWindows, 25) AS STRING
    COMMON SHARED /WindowA/ Rows() AS INTEGER, Cols() AS INTEGER,⌉
                  Toprow() AS INTEGER, TopCol() AS INTEGER,⌉
                  BotRow() AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /WindowB/ Attribute() AS INTEGER,⌉
                  OnFlag() AS INTEGER, _
                 Text() AS STRING, OldText() AS STRING,⌉
                                   OldAttrb() AS STRING

SUB WindowHide (Handle%)

    CurRow% = CSRLIN
    CurCol% = POS(0)

    DIM InRegs AS RegType, OutRegs AS RegType

    REM Stampa il testo precedente

    LOCATE TR, TC
    InRegs.cx = 1
```

continua

Listato 2.3 *Sottoprogramma WindowHide per rimuovere le finestre.*

```
Sottoprogramma WindowHide per nascondere le finestre.

    FOR i = 1 TO Rows(Handle%)
       FOR j = 1 TO Cols(Handle%)
          InRegs.ax = &H900 + ASC(MID$(OldText(Handle%, i), j, 1))
          InRegs.bx = ASC(MID$(OldAttrb(Handle%, i), j, 1))
          CALL INTERRUPT(&H10, InRegs, OutRegs)
          LOCATE TR + i - 1, TC  + j
       NEXT j
          LOCATE TR + i, TC
    NEXT i

    REM Disattivazione visualizzazione finestra

    OnFlag(Handle%) = 0

    LOCATE CurRow%, CurCol%

END SUB
```

Listato 2.3 *Sottoprogramma WindowHide per rimuovere le finestre.*

Anche il questo caso il listato non è poi così breve ma, indubbiamente, è di dimensioni inferiori rispetto a *WindowShow( )*. La differenza di estensione è causata dal fatto che *WindowHide( )* ha solo bisogno di ripristinare il testo e i relativi attributi precedentemente visualizzati e non necessita di alcun tipo di memorizzazione della finestra prima di visualizzarla poiché, in questo caso, il testo è già presente nelle istruzioni *COMMON* della finestra. Ora si vedrà come operare con la funzione *WindowHide( )*.

PROVA DI RIMOZIONE DI UNA FINESTRA

Nell'esempio presentato in questo paragrafo, verranno utilizzate le finestre già create con la funzione *WindowsShow( )* e ne verrà fatta scomparire solo una. Prima di tutto, si farà comparire a video la finestra 1 e in seguito ad essa verrà sovrapposta la finestra 2 che ne coprirà completamente il contenuto. Successivamente, la finestra 2 verrà rimossa consentendo nuovamente la visualizzazione del contenuto della finestra 1.

```
DECLARE FUNCTION WindowGetHandle% (TopRow, TopCol%, NumRow%,
                                   NumCol%, Attr%)
DECLARE FUNCTION WindowPrint% (Handle%, Row%, Col%, Pstring$)
DECLARE SUB WindowShow (Handle%)
DECLARE SUB WindowHide (Handle%)

Handle1% = WindowGetHandle%(10, 10, 5, 5, &H24)
```

```
Handle2% = WindowGetHandle%(19, 10, 3, 13, &H61)

CALL WindowShow(Handle1%)
CALL WindowShow(Handle2%)
CALL WindowHide(Handle2%)
```

L'uso di *WindowShow( )* e *WindowHide( )* mostra la versatilità dei sottoprogrammi. Per visualizzare una finestra sarà sufficiente utilizzare *WindowShow( )* mentre, per rimuoverla, si dovrà utilizzare *WindowHide( )*. Da quanto sopra si vede come è semplice, in fase di programmazione, richiamare queste funzioni. Quando saranno stati messi a punto tutti i dettagli, queste funzioni potranno essere utilizzate in un qualsiasi programma.
A questo punto le finestre sono state inizializzate, le si sono fatte apparire o scomparire e, in sostanza, si sono fatti molti passi avanti. D'altra parte è anche vero che una finestra che non contiene alcun tipo di testo non serve proprio a nulla, perciò ora si vedrà come visualizzare il testo che viene immesso in una finestra.

STAMPA DI UN TESTO IN UNA FINESTRA

Alle finestre che sono state appena create, si dovranno ora aggiungere una o più righe di testo sviluppando un sottoprogramma a cui si assegnerà il nome di *WindowPrint( )* che è in grado di visualizzare del testo nelle finestre. Il processo è relativamente semplice: quello che è davvero necessario è la definizione della posizione all'interno della finestra alla quale inizierà la visualizzazione del testo (dato per *[1, 1]* l'angolo superiore sinistro della finestra) e il contenuto del testo stesso.
Poiché una stringa potrebbe non sempre essere contenuta completamente in una finestra, il sottoprogramma dovrebbe anche prevedere determinate indicazioni della corretta o mancata immissione del testo.
Innanzitutto si cambi *WindowPrint( )* nella funzione *WindowPrint%( )*: se la funzione sarà in grado di trasferire la stringa alla finestra, verrà restituito il valore 1, in caso contrario, il valore restituito sarà 0.

> ***Consiglio:*** In generale, se si prevede che il sottoprogramma non possa essere eseguito correttamente, sarebbe meglio trasformarlo in funzione poiché questa sarà in grado di restituire un valore di corretta o scorretta esecuzione.

Tenendo presente quanto sopra, di seguito viene mostrato come si dovrebbe utilizzare *WindowPrint%( )*.

```
IF WindowPrint% (Handle%, PrtnRow%, PrntCol%, PrntStr$) THEN
   PRINT "Stringa immessa correttamente nella finestra."
```

```
ELSE
   PRINT "ERRORE!"
   PRINT "La finestra non è in grado di contenere la stringa."
END IF
```

***Nota**: Verranno utilizzate sia WindowPrint%( ) sia il controllo del valore che da essa viene restituito: questa tecnica è, in sostanza, una reminiscenza della programmazione in linguaggio C.*

Si osservi ora il contenuto di alcuni elementi del codice che verrà generato:

Nome parametro	*Tipo*	*Descrizione*
Handle%	INTEGER	Il gestore di questa finestra viene restituito dalla funzione *WindowGet Handle%( )*. I gestori validi sono quelli diversi da zero e contenuti nell'intervallo da 1 a 8.
PrntRow%	INTEGER	Corrisponde alla riga all'interno della finestra alla quale inizierà la visualizzazione della stringa. Utilizzare le coordinate locali della finestra. Per esempio, l'angolo superiore sinistro sarà pari a *(1, 1)*.
PrntCol%	INTEGER	Indica la colonna all'interno della finestra alla quale inizierà la visualizzazione della stringa. Utilizzare le coordinate locali della finestra. Per esempio, l'angolo superiore della finestra corrisponde a *(1, 1)*.
PrntStr$	STRING	Contenuto della stringa che verrà visualizzata all'interno della finestra a partire dalla posizione indicata in *PrntRow%* e *PrntCol%*.

La stringa potrà essere trasferita in modo che questa venga emessa a video come un stringa normale. Per esempio, la stringa "E' arrivato il momento.", viene generata dal frammento di seguito riportato:

```
PrntRow% = i
PrntCol% = 1
PrntStr$ = "E' arrivato il momento."
IF WindowPrint% (Handle%, PrntRow%, PrntCol%, PrntStr$) THEN
   PRINT "La stringa è stata immessa correttamente nella finestra."
ELSE
   PRINT "ERRORE!"
```

```
    PRINT "Non è possibile immettere la stringa nella finestra."
  END IF
```

Il frammento sopra riportato genera l'emissione riportata in figura 2.8.

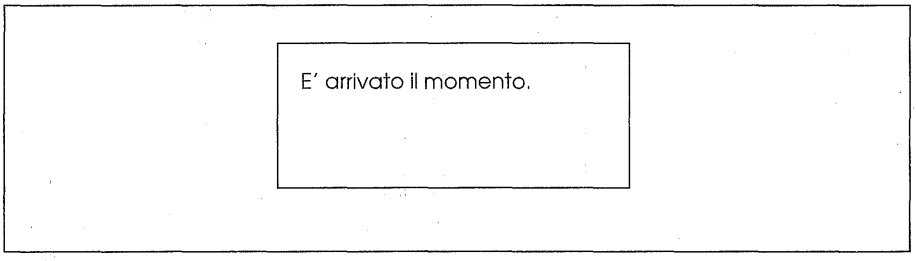

Figura 2.8

Si tenga presente che se la stringa è troppo lunga per essere contenuta all'interno della riga definita per la finestra, sarà necessario suddividerla in modo che la parte eccedente venga portata sulla riga successiva, ma come si può materialmente far slittare alla riga successiva la parte eccedente della stringa che non può essere contenuta in una sola riga? In questo caso sarà necessario avviare alcune operazioni che inseriscano un ritorno di carrello all'interno della stringa che dovrà essere emessa a video. Innanzitutto è necessario controllare che all'interno della stringa sia stato immesso *CHR$(13)*, corrispondente a un ritorno di carrello, in modo da consentire a *WindowPrint%( )* di portare il cursore alla prima posizione della riga successiva. Per esempio, per suddividere su due righe il messaggio sopra riportato, e per ottenere un'emissione simile a quella riportata in figura 2.9, sarà necessario trasferire a *WindowPrint%* la stringa "E' arrivato il"+CHR$(13)+"momento."
Inoltre si noti che, poiché si sta lavorando su una finestra che non è stata riportata a video, la visualizzazione della stringa non viene immediatamente riportata a video.

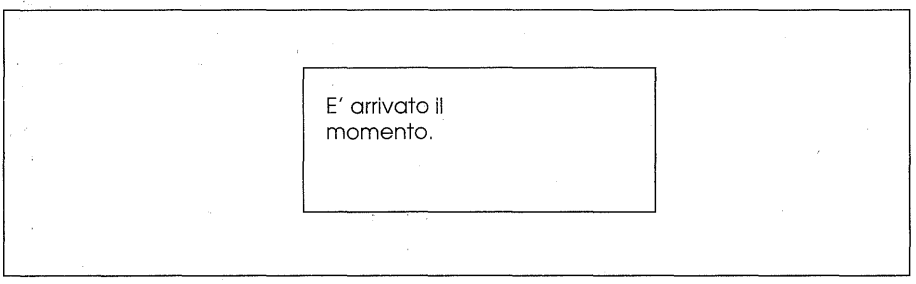

Figura 2.9

Al contrario, se la finestra è *già* a video, *WindowPrint%( )* aggiornerà la stringa precedentemente visualizzata all'interno della finestra.
Si proverà ora a creare questa funzione. Innanzitutto dovrà essere realizzata una copia non distruttiva dei parametri che verranno trasferiti: *PrntRow%, PrntCol%* e *PrntStr$*; si dovrà anche salvare la posizione del cursore in quanto *WindowPrint%( )* dovrà in seguito aggiornare l'emissione a video.

```
FUNCTION WindowPrint% (Handle%, PrntRow%, PrntCol%, PrntStr$)

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   WindowPrint% = 1

   REM Esegue copie non distruttive

   PrintRow% = PrntRow%
   PrintCol% = PrntCol%
   PrintString$ = PrntStr$
   :
```

Si è così ottimisticamente assegnata a *WindowPrint%* una restituzione di valore pari a 1 — che indica la corretta esecuzione dell'operazione — e si è impostato un loop sulle righe della finestra durante il quale dovranno essere inseriti i caratteri della stringa.

> ***Consiglio***: In programmazione, la corretta esecuzione di un'operazione viene indicata da valori non negativi.

Il loop di cui sopra richiede una breve analisi. L'idea di partenza è quella di immettere i caratteri nella stringa della finestra senza che venga superato il limite imposto dal numero di righe che la finestra è in grado di contenere (per esempio, *PrintRow%* <= *Rows(Handle%)*) e che vengano immessi tutti i caratteri fino a quando non verrà verificata l'assenza di ulteriori caratteri (*PrintString$* <>"").

```
FUNCTION WindowPrint% (Handle%, PrntRow%, PrntCol%, PrntStr$)

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   WindowPrint% = 1
```

```
REM Esegue copie non distruttive

PrintRow% = PrntRow%
PrintCol% = PrntCol%
PrintString$ = PrntStr$

DO
:
:
   LOOP WHILE (PrintRow% <= Rows(Handle%)) AND (PrintString$ <> "")
:
:
```

In seguito dovrà essere immesso un loop interno che agirà sul numero di colonne della finestra. In effetti, nella finestra potranno essere immessi caratteri fintanto che non verrà raggiunto il margine destro della finestra stessa (per esempio: *PrintCol% <= Cols(Handle%)*) e fintanto che vi saranno caratteri da inserirvi (*PrintString$ <> ""*).

```
FUNCTION WindowPrint% (Handle%, PrntRow%, PrntCol%, PrntStr$)

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   WindowPrint% = 1

   REM Esegue copie non distruttive

   PrintRow% = PrntRow%
   PrintCol% = PrntCol%
   PrintString$ = PrntStr$
   DO
      DO
         :
         [Immette i caratteri in una finestra]
         :
         PrintCol% = PrintCol% + 1
      LOOP WHILE (PrintCol% <= Cols(Handle%))
                 AND (PrintString$ <> "")

      PrintRow% = PrintRow% + 1
      PrintCol% = 1

   LOOP WHILE (PrintRow% <= Rows(Handle%)) AND (PrintString$ <> "")
   :
   :
```

Tenere presente che sarà sempre possibile incrementare la colonna alla fine di ciascuna interazione sulle colonne e incrementare il numero di righe (impostando nel frattempo in numero di colonna nuovamente a 1) dopo ciascuna iterazione su ciascuna riga. Questi due loop consentono di avviare un ciclo sulla riga corrente della finestra (figura 2.10) e, quando si sarà raggiunta la fine di una riga, di passare alla riga successiva (figura 2.11).

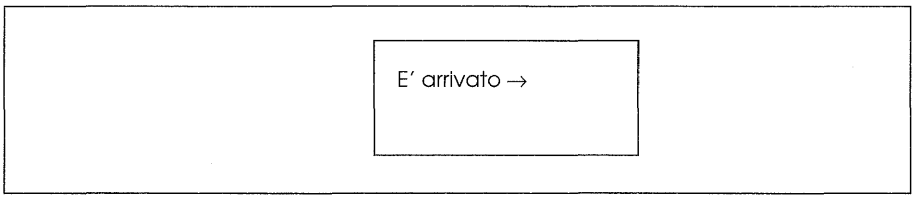

Figura 2.10

Figura 2.11

A questo punto si sarà pronti al trasferimento dei caratteri della stringa prelevandoli da *PrintString$* e immettendoli, tramite *PrintRow%* e *PrintCol%*, all'interno della finestra (si tenga presente che il testo della finestra è contenuto in un array a cui è stato assegnato il nome *Text( )*).

```
FUNCTION WindowPrint% (Handle%, PrntRow%, PrntCol%, PrntStr$)

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   WindowPrint% = 1

   REM Esegue copie non distruttive

   PrintRow% = PrntRow%
   PrintCol% = PrntCol%
```

```
PrintString$ = PrntStr$

DO

   DO
      Char$ = LEFT$(PrintString$, 1)
      PrintString$ = RIGHT$(PrintString$, LEN(PrintString$) - 1)
      IF ASC(Char$) = 13 THEN EXIT DO
      MID$(Text(Handle%, PrintRow%), PrintCol%, 1) = Char$
      PrintCol% = PrintCol% + 1
   LOOP WHILE (PrintCol% <= Cols(Handle%))
              AND (PrintString$ <> "")

   PrintRow% = PrintRow% + 1
   PrintCol% = 1

LOOP WHILE (PrintRow% <= Rows(Handle%)) AND (PrintString$ <> "")
:
:
```

In questo loop, in effetti, vengono inseriti nella finestra i caratteri richiesti. La copia distruttiva della stringa da visualizzare è contenuta in *PrintString$* e il carattere emesso sarà quello relativo alla posizione corrente definita da *LEFT$(PrintString$, 1)*, prelevato da *PrintString$* e inserito nella riga corretta in *Text(Handle%)*.

> ***Consiglio***: Un semplice metodo per prelevare la prima parola di una stringa (e non solo il primo carattere) è quello di utilizzare la funzione *INSTR( )* nella seguente forma **`FirwstWord$ = LEFT$(MyString$, INSTR(MyString$, " ")-1)`**.

Il loop continuerà su tutte le colonne fino a quando non verrà raggiunto il bordo destro della finestra. Si noti, inoltre, che è necessario controllare la presenza all'interno della stringa, di caratteri corrispondenti a un ritorno di carrello, poiché questi generano la creazione di una nuova riga. Dopo che sarà terminata l'immissione dei caratteri in ogni riga, viene avviato il loop che controlla la presenza di righe successive fino a quando non ve ne saranno più di disponibili o fino a quando non verrà raggiunto l'ultimo carattere immesso.

Al termine di questi due cicli, la stringa dovrebbe essere stata completamente immessa, oppure si saranno raggiunti i limiti fisici della finestra (sarà necessario, in ogni caso, controllare quale delle due condizioni si è avverata).

Se si sono raggiunti i limiti fisici della finestra, sarà necessario generare la restituzione di un valore *0* che indica che la finestra non è in grado di contenere completamente la stringa di testo.

```
FUNCTION WindowPrint% (Handle%, PrntRow%, PrntCol%, PrntStr$)

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   WindowPrint% = 1

   REM Esegue copie non distruttive

   PrintRow% = PrntRow%
   PrintCol% = PrntCol%
   PrintString$ = PrntStr$

   DO

      DO
         Char$ = LEFT$(PrintString$, 1)
         PrintString$ = RIGHT$(PrintString$, LEN(PrintString$) - 1)
         IF ASC(Char$) = 13 THEN EXIT DO
         MID$(Text(Handle%, PrintRow%), PrintCol%, 1) = Char$
         PrintCol% = PrintCol% + 1
      LOOP WHILE (PrintCol% <= Cols(Handle%))
                 AND (PrintString$ <> "")

      PrintRow% = PrintRow% + 1
      PrintCol% = 1

   LOOP WHILE (PrintRow% <= Rows(Handle%)) AND (PrintString$ <> "")

   IF PrintString$ <> "" THEN WindowPrint% = 0
   :
   :
```

In caso contrario, *WindowPrint* dovrà essere impostato a *1*, che indica la corretta immissione di tutti i caratteri della stringa. L'ultima operazione da eseguire è il controllo della visualizzazione della finestra (ossia controllare se *OnFlag(Handle%)* è pari a *1*), nel qual caso sarà necessario aggiornare il contenuto del video.
Per eseguire questo aggiornamento si potrà ridisegnare nuovamente la finestra utilizzando *WindowShow( )*. Il listato 2.4 mostra la composizione completa della funzione *WindowPrint%()*.

```
DECLARE FUNCTION WindowPrint% (Handle%, PrntRow%, PrntCol%,⌉
                               PrntStr$)

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,⌉
                       OutRegs AS RegType)

    CONST MaxWindows = 8
    DIM Rows(1 TO MaxWindows) AS INTEGER
    DIM Cols(1 TO MaxWindows) AS INTEGER
    DIM TopRow(1 TO MaxWindows) AS INTEGER
    DIM TopCol(1 TO MaxWindows) AS INTEGER
    DIM BotRow(1 TO MaxWindows) AS INTEGER
    DIM BotCol(1 TO MaxWindows) AS INTEGER
    DIM Attribute(1 TO MaxWindows) AS INTEGER
    DIM OnFlag(1 TO MaxWindows) AS INTEGER
    DIM Text(MaxWindows, 25) AS STRING
    DIM OldText(MaxWindows, 25) AS STRING
    DIM OldAttrb(MaxWindows, 25) AS STRING
    COMMON SHARED /WindowA/ Rows() AS INTEGER, Cols() AS INTEGER,⌉
                  Toprow() AS INTEGER, TopCol() AS INTEGER,⌉
                  BotRow() AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /WindowB/ Attribute() AS INTEGER,⌉
                  OnFlag() AS INTEGER,⌉
                  Text() AS STRING, OldText() AS STRING,⌉
                  OldAttrb() AS STRING

FUNCTION WindowPrint% (Handle%, PrntRow%, PrntCol%, PrntStr$)

    DIM InRegs AS RegType, OutRegs AS RegType
    CurRow% = CSRLIN
    CurCol% = POS(0)

    WindowPrint% = 1
    REM Esegue copie non distruttive
```

continua

Listato 2.4 *Funzione WindowPrint% per visualizzare il testo in una finestra.*

```
    PrintRow% = PrntRow%
    PrintCol% = PrntCol%
    PrintString$ = PrntStr$

    DO
      DO
         Char$ = LEFT$(PrintString$, 1)
         PrintString$ = RIGHT$(PrintString$,
                        LEN(PrintString$) - 1)
         IF ASC(Char$) = 13 THEN EXIT DO
         MID$(Text(Handle%, PrintRow%), PrintCol%, 1) = Char$
         PrintCol% = PrintCol% + 1
      LOOP WHILE (PrintCol% <= Cols(Handle%))
                 AND (PrintString$ <> "")

      PrintRow% = PrintRow% + 1
      PrintCol% = 1

    LOOP WHILE (PrintRow% <= Rows(Handle%))
               AND (PrintString$ <> "")

    IF PrintString$ <> "" THEN WindowPrint% = 0

    IF OnFlag(Handle%) THEN

      REM Visualizza il nuovo testo
      TR = TopRow(Handle%)
      TC = TopCol(Handle%)

      LOCATE TR, TC
      InRegs.cx = 1

      FOR i = 1 TO Rows(Handle%)
         FOR j = 1 TO Cols(Handle%)
            InRegs.ax = &H900 + ASC(MID$(Text(Handle%, i), j, 1))
            InRegs.bx = Attribute(Handle%)
            CALL INTERRUPT(&H10, InRegs, OutRegs)
            LOCATE TR + i - 1, TC  + j
         NEXT j
            LOCATE TR + i, TC
      NEXT i

    END IF
    LOCATE CurRow%, CurCol%
END FUNCTION
```

Listato 2.4 *Funzione WindowPrint% per visualizzare il testo in una finestra.*

Anche questo è un listato abbastanza lungo poiché prevede l'aggiornamento della finestra nel caso in cui questa sia visibile a video — in effetti, il vero scopo della creazione di una finestra è quello di immettervi un numero di righe relativamente elevato. Ora si cercherà di operare con la funzione *WindowPrint%( )*.

> ***Consiglio***: La funzione *WindowPrint%( )* potrà anche essere utilizzata per stampare i bordi di una finestra, nel caso in cui vengano utilizzati caratteri ASCII di tipo grafico.

PROVA DI STAMPA DI UN TESTO IN UNA FINESTRA

Di seguito viene riportato un semplice programma che controlla la corretta esecuzione di *WindowPrint%( )*. In questo esempio verrà semplicemente visualizzata la stringa "Salve!" a partire dalla posizione (1, 1) e la stringa "Test" a partire dalla posizione (5, 1).

```
DECLARE FUNCTION WindowGetHandle% (TopRow%, TopCol%, NumRow%,
                                   NumCol%, Attr%)
DECLARE SUB WindowShow (Handle%)
DECLARE FUNCTION WindowPrint% (Handle%, Row%, Col%, PrintString$)

Handle% = WindowGetHandle% (10, 10, 5, 5, &H24)

CALL WindowShow(Handle%)

Check% = WindowPrint%(Handle%, 1, 1, "Salve!")

Check% = WindowPrint%(Handle%, 5, 1, "Test")

IF Check% = 0 THEN PRINT "Errore!"
```

Nota: *Le coordinate di riga e colonna che vengono trasferite a WindowPrint%( ) sono calcolate in relazione alla finestra: vale a dire che (1, 1) corrisponde all'angolo superiore sinistro della finestra.*

Il risultato di questo programma sarà una finestra di piccole dimensioni simile a quella riportata in figura 2.12.
Ora che si è in grado di impostare una finestra, visualizzarla, immettervi un testo e rimuoverla si è indubbiamente compiuto più di un passo in avanti. Tuttavia vi è un'ulteriore opzione che non è ancora stata presa in esame: lo spostamento delle finestre all'interno dello schermo. Per esempio, se si desidera spostare una finestra dal centro dello schermo a una determinata posizione dello stesso sarà necessario creare una funzione a cui assegnare il nome *WindowMove( )*.

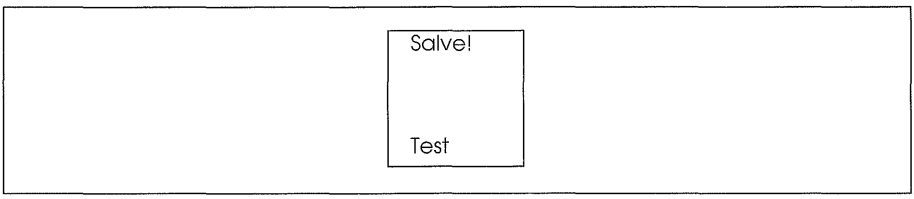

Figura 2.12

SPOSTAMENTO DI FINESTRE

Anche se la funzione *WindowGetHandle%( )* è in grado di impostare la posizione iniziale della finestra al momento della sua creazione, non sarà sempre obbligatorio mantenere fissa questa posizione, anzi, molte volte le finestre dovranno essere spostate altrove. In questo caso è necessario generare una funzione a cui assegnare il nome *WindowMove%( )* in grado di spostare le finestre nella posizione più appropriata. Poiché lo spostamento di una finestra potrebbe anche non essere possibile — per esempio la finestra potrebbe "sbordare" dai limiti fisici imposti dallo schermo — questa funzione dovrebbe restituire un valore pari a *1* in caso di spostamento corretto e un valore pari a *0* in caso di errore.
Come accade per *WindowPrint%( )*, anche la funzione *WindowMove%( )* dovrebbe mantenere invisibile la finestra; tuttavia, nel caso in cui una finestra sia *già presente* a video, lo schermo dovrà essere aggiornato facendo scomparire dalla sua precedente posizione la finestra su cui si sta operando e facendola riapparire nella sua nuova posizione. In altre parole, se la finestra è già a video, e si desidera spostarla, sarà necessario eseguire i seguenti passi:

1. ripristinare l'area dello schermo occupata dalla finestra che si intende spostare;
2. salvare la posizione di destinazione della finestra;
3. richiamare a video la finestra nella sua nuova posizione.

I tre passi sopra indicati genereranno una funzione *WindowMove%( )* abbastanza lunga ma si tenga presente che sono già state scritte le procedure per le funzioni *WindowShow( )* e *WindowHide( )*.

Consiglio: L'utilizzo di routine personalizzate rende molto più rapida l'operazione di programmazione; in questa fase si tenga ben presente di utilizzare, dove necessario, le routine già realizzate, in quanto questa procedura consente di velocizzare le operazioni di debug dell'intero codice.

Per utilizzare la funzione *WindowMove%( )* per spostare la finestra in una nuova posizione (per esempio per spostarla all'angolo superiore sinistro dello schermo) sarà necessario fornirle l'*identificatore* della finestra. Se si desidera spostare una finestra nell'angolo superiore sinistro dello schermo, sarà necessario scrivere il frammento di codice di seguito riportato:

```
NewTorRow% = 1
NewTopCol% = 1
IF WindowMove% (Handle%, NewTopRow%, NewTopCol%) THEN
   PRINT "Finestra spostata correttamente."
ELSE
   PRINT "Errore! Impossibile spostare la finestra."
END IF
```

Osservare ora il contenuto generale di *WindowMove%( )*:

Nome parametro	*Tipo*	*Descrizione*
Handle%	INTEGER	Il gestore di questa finestra viene restituito dalla funzione *WindowGetHandle%( )*. I gestori validi sono quelli diversi da zero e contenuti nell'intervallo da 1 a 8.
NewTopRow%	INTEGER	Corrisponde alla nuova riga dello schermo (da 1 a 24) dell'angolo superiore sinistro della finestra che deve essere spostata.
NewTopCol%	INTEGER	Indica la nuova colonna (da 1 a 79) dell'angolo superiore sinistro a cui spostare l'angolo superiore sinistro della finestra.

Si passerà ora allo sviluppo del programma. Innanzitutto si dovranno avviare alcuni controlli di limiti (per verificare che la finestra non fuoriesca dal limiti fisici dello schermo):

```
WindowMove% = 1
IF NewTopRow% < 1 OR NewTopCol% < 1 THEN
   WindowMove% = 0
   EXIT FUNCTION
END IF
IF NewTopRow% + Rows(Handle%) > 25 OR NewTopCol% + Cols(Handle%)
        > 80 THEN
   WindowMove% = 0
   EXIT FUNCTION
END IF
:
```

Se si è ancora all'interno della funzione, lo spostamento risulta accettabile. Per spostare la finestra sarà sufficiente compilare nuovamente gli array relativi alle posizioni delle istruzioni *COMMON* della finestra: *TopRow(Handle%)*, *TopCol(Handle%)*, *BotRow(Handle%)* e *BotCol(Handle%)* (figura 2.13).

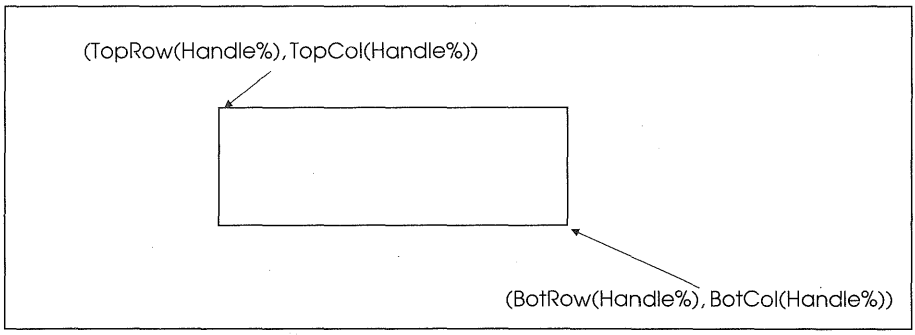

Figura 2.13

Per effettuare questa operazione si proceda come di seguito riportato:

```
WindowMove% = 1

IF NewTopRow% < 1 OR NewTopCol% < 1 THEN
   WindowMove% = 0
   EXIT FUNCTION
END IF

IF NewTopRow% + Rows(Handle%) > 25 OR NewTopCol%
        + Cols(Handle%) > 80 THEN
   WindowMove% = 0
   EXIT FUNCTION
END IF

TopRow(Handle%) = NewTopRow%
TopCol(Handle%) = NewTopCol%
BotRow(Handle%) = NewTopRow% + Rows(Handle%)
BotCol(Handle%) = NewTopCol% + Cols(Handle%)

IF OnFlag(Handle%) = 0 THEN EXIT FUNCTION
:
:
```

Nel caso in cui la finestra non sia visibile a video, l'operazione è terminata e si uscirà dalla funzione. In caso contrario, sarà necessario eseguire ancora ulteriori passi. In effetti, lo scopo principale di questo programma è quello di spostare la finestra, ma

bisogna opera diversamente se questa è visibile (ovvero quando *OnFlag(Handle%)* è impostato a 1).
In questo caso sarà prima necessario nascondere la finestra che dovrà essere spostata ripristinando nel contempo il testo in essa contenuto prima della sua visualizzazione. Si potrà in questo caso adattare il codice contenuto in *WindowHide( )* in modo che questa operazione venga eseguita in modo automatico. Innanzitutto sarà necessario conservare la posizione originale della finestra prima di aggiornarne le coordinate e ripristinare il testo all'interno di essa:

```
DIM InRegs AS RegType, OutRegs AS RegType

CurRow% = CSRLIN
CurCol% = POS(0)

WindowMove% = 1

IF NewTopRow% < 1 OR NewTopCol% < 1 THEN
   WindowMove% = 0
   EXIT FUNCTION
END IF

IF NewTopRow% + Rows(Handle%) > 25 OR NewTopCol% +
        Cols(Handle%) > 80 THEN
   WindowMove% = 0
   EXIT FUNCTION
END IF

OldTopRow = TopRow(Handle%)
OldTopCol = TopCol(Handle%)
TopRow(Handle%) = NewTopRow%
TopCol(Handle%) = NewTopCol%
BotRow(Handle%) = NewTopRow% + Rows(Handle%)
BotCol(Handle%) = NewTopCol% + Cols(Handle%)

IF OnFlag(Handle%) = 0 THEN EXIT FUNCTION

REM Ripristina la precedente area dello schermo

TR = OldTopRow
TC = OldTopCol
LOCATE TR, TC
InRegs.cx = 1

FOR i = 1 TO Rows(Handle%)
   FOR j = 1 TO Cols(Handle%)
      InRegs.ax = &H900 + ASC(MID$(OldText(Handle%, i), j, 1))
      InRegs.bx = ASC(MID$(OldAttrb(Handle%, i), j, 1))
      CALL INTERRUPT(&H10, InRegs, OutRegs)
```

```
        LOCATE TR + i - 1, TC + j
      NEXT j
        LOCATE TR + i, TC
    NEXT i
    :
    :
  END SUB
```

In seguito, sarà necessario spostare la finestra. Si inizierà con la memorizzazione dell'area di destinazione in cui comparirà la finestra utilizzando il codice di *WindowShow( )*:

```
DIM InRegs AS RegType, OutRegs AS RegType

CurRow% = CSRLIN
CurCol% = POS(0)

WindowMove% = 1

IF NewTopRow% < 1 OR NewTopCol% < 1 THEN
   WindowMove% = 0
   EXIT FUNCTION
END IF

IF NewTopRow% + Rows(Handle%) > 25 OR NewTopCol% +
      Cols(Handle%) > 80 THEN
   WindowMove% = 0
   EXIT FUNCTION
END IF

OldTopRow = TopRow(Handle%)
OldTopCol = TopCol(Handle%)
TopRow(Handle%) = NewTopRow%
TopCol(Handle%) = NewTopCol%
BotRow(Handle%) = NewTopRow% + Rows(Handle%)
BotCol(Handle%) = NewTopCol% + Cols(Handle%)

IF OnFlag(Handle%) = 0 THEN EXIT FUNCTION

REM Ripristina la precedente area dello schermo

TR = OldTopRow
TC = OldTopCol
LOCATE TR, TC
InRegs.cx = 1

FOR i = 1 TO Rows(Handle%)
   FOR j = 1 TO Cols(Handle%)
      InRegs.ax = &H900 + ASC(MID$(OldText(Handle%, i), j, 1))
```

```
        InRegs.bx = ASC(MID$(OldAttrb(Handle%, i), j, 1))
        CALL INTERRUPT(&H10, InRegs, OutRegs)
        LOCATE TR + i - 1, TC + j
     NEXT j
        LOCATE TR + i, TC
  NEXT i

  REM Salva l'area di destinazione dello schermo

  TR = TopRow(Handle%)
  TC = TopCol(Handle%)

  FOR i = 1 TO Rows(Handle%)
     temp1$ = ""
     temp2$ = ""
     FOR j = 1 TO Cols(Handle%)
        temp1$ = temp1$ + CHR$(SCREEN(TR + i - 1, TC + j - 1))
        temp2$ = temp2$ + CHR$(SCREEN(TR + i - 1, TC + j - 1, 1))
     NEXT j
     OldText(Handle%, i) = temp1$
     OldAttrb(Handle%, i) = temp2$
  NEXT i
  :
  :
```

Ora sarà possibile visualizzare la finestra a video utilizzando nuovamente il codice prelevato da *WindowShow( )*. Il listato 2.5 mostra la funzione completa.

```
DECLARE FUNCTION WindowMove% (Handle%, NewTopRow%, NewTopCol%)

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)
```

continua

Listato 2.5 *La funzione WindowMove% per lo spostamento di una finestra all'interno dello schermo.*

```
    CONST MaxWindows = 8
    DIM Rows(1 TO MaxWindows) AS INTEGER
    DIM Cols(1 TO MaxWindows) AS INTEGER
    DIM TopRow(1 TO MaxWindows) AS INTEGER
    DIM TopCol(1 TO MaxWindows) AS INTEGER
    DIM BotRow(1 TO MaxWindows) AS INTEGER
    DIM BotCol(1 TO MaxWindows) AS INTEGER
    DIM Attribute(1 TO MaxWindows) AS INTEGER
    DIM OnFlag(1 TO MaxWindows) AS INTEGER
    DIM Text(MaxWindows, 25) AS STRING
    DIM OldText(MaxWindows, 25) AS STRING
    DIM OldAttrb(MaxWindows, 25) AS STRING
    COMMON SHARED /WindowA/ Rows() AS INTEGER, Cols() AS INTEGER,⌉
       TopRow() AS INTEGER, TopCol() AS INTEGER,⌉
       BotRow() AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /WindowB/ Attribute() AS INTEGER,⌉
       OnFlag() AS INTEGER, Text() AS STRING,⌉
       OldText() AS STRING, OldAttrb() AS STRING

FUNCTION WindowMove% (Handle%, NewTopRow%, NewTopCol%)

    DIM InRegs AS RegType, OutRegs AS RegType
    CurRow% = CSRLIN
    CurCol% = POS(0)
    WindowMove% = 1

    IF NewTopRow% < 1 OR NewTopCol% < 1 THEN
       WindowMove% = 0
       EXIT FUNCTION
    END IF

    IF NewTopRow% + Rows(Handle%) > 25 OR NewTopCol% +⌉
          Cols(Handle%) > 80 THEN
       WindowMove% = 0
       EXIT FUNCTION
    END IF

    OldTopRow = TopRow(Handle%)
    OldTopCol = TopCol(Handle%)
    TopRow(Handle%) = NewTopRow%
    TopCol(Handle%) = NewTopCol%
    BotRow(Handle%) = NewTopRow% + Rows(Handle%)
    BotCol(Handle%) = NewTopCol% + Cols(Handle%)

    IF OnFlag(Handle%) = 0 THEN EXIT FUNCTION
```

continua

Listato 2.5 *La funzione WindowMove% per lo spostamento di una finestra all'interno dello schermo.*

```
    REM Ripristina la precedente area dello schermo
    TR = OldTopRow
    TC = OldTopCol
    LOCATE TR, TC
    InRegs.cx = 1

    FOR i = 1 TO Rows(Handle%)
       FOR j = 1 TO Cols(Handle%)
          InRegs.ax = &H900 + ASC(MID$(OldText(Handle%, i), j, 1))
          InRegs.bx = ASC(MID$(OldAttrb(Handle%, i), j, 1))
          CALL INTERRUPT(&H10, InRegs, OutRegs)
          LOCATE TR + i - 1, TC + j
       NEXT j
          LOCATE TR + i, TC
    NEXT i

    REM Salva l'area di destinazione dello schermo
    TR = TopRow(Handle%)
    TC = TopCol(Handle%)

    FOR i = 1 TO Rows(Handle%)
       temp1$ = ""
       temp2$ = ""
       FOR j = 1 TO Cols(Handle%)
          temp1$ = temp1$ + CHR$(SCREEN(TR + i - 1, TC + j - 1))
          temp2$ = temp2$ + CHR$(SCREEN(TR + i - 1, TC + j - 1, 1))
       NEXT j
       OldText(Handle%, i) = temp1$
       OldAttrb(Handle%, i) = temp2$
    NEXT i

    REM Stampa il nuovo testo
       LOCATE TR, TC
       InRegs.cx = 1
       FOR i = 1 TO Rows(Handle%)
          FOR j = 1 TO Cols(Handle%)
             InRegs.ax = &H900 + ASC(MID$(Text(Handle%, i), j, 1))
             InRegs.bx = Attribute(Handle%)
             CALL INTERRUPT(&H10, InRegs, OutRegs)
             LOCATE TR + i - 1, TC + j
          NEXT j
          LOCATE TR + i, TC
       NEXT i
       LOCATE CurRow%, CurCol%
END FUNCTION
```

Listato 2.5 *La funzione WindowMove% per lo spostamento di una finestra all'interno dello schermo.*

E con questo programma si è ora in grado di spostare le finestre all'interno dello schermo. Si provi ora ad avviare il programma.

PROVA DI SPOSTAMENTO DI FINESTRE

Viene qui presentato un esempio di programma che mostra l'uso di *WindowMove%( )*.

```
DECLARE FUNCTION WindowGetHandle% (TopRow%, TopCol%,
                                   NumRow%, NumCol%, Attr%)
DECLARE SUB WindowShow (Handle%)
DECLARE FUNCTION WindowMove% (Handle% NewTopRow%, NewTopCol%)

Handle1% = WindowGetHandle%(10, 10, 5, 5, &H24)
Handle2% = WindowGetHandle(20, 20, 3, 13, &H61)

CALL WindowShow(Handle1%)
CALL WindowShow(Handle2%)

PRINT "Premere un tasto per spostare la finestra 2."
DO
LOOP WHILE INKEY$ = " "

Check% = WindowMove%(Handle2%, 5, 20)

IF Check% = 0 THEN PRINT "Errore!"
```

In questo esempio a video sono state immesse due finestre e in seguito il programma attende la pressione di un tasto. Quando il programma leggerà il tasto immesso, la finestra 2 verrà spostata dalle coordinate *(20, 20)* alle coordinate *(5, 20)* (si veda figura 2.13). Si ricorda che queste coordinate sono coordinate video relative all'angolo superiore sinistro della finestra da spostare.
Si è conclusa a questo punto la creazione di una serie di funzioni e sottoprogrammi destinati alla creazione, impostazione, visualizzazione e spostamento di finestre utilizzabili in un qualsiasi altro programma. L'unica funzione che ora manca è quella relativa alla cancellazione di finestre precedentemente create.

CANCELLAZIONE DI UNA FINESTRA

Il sottoprogramma finale di questo capitolo prenderà in esame la procedura per cancellare una finestra precedentemente creata e liberarne, di conseguenza, il relativo identificatore. In effetti, l'unico elemento che è necessario è l'identificatore

delle finestre; per cancellare una finestra si dovrà utilizzare una funzione a cui verrà assegnato il nome *WindowDelete(Handle%)*. Questa funzione è particolarmente importante quando si è raggiunto il numero massimo di finestre in memoria e quando si presume di doverne aggiungere altre in una successiva parte del programma.

***Nota**: La cancellazione di una finestra è un'operazione diversa da quella avviata per rimuoverla. Quando si sarà eliminato l'assegnamento di un identificatore di finestra, non sarà più possibile visualizzare nuovamente la finestra ad esso precedentemente assegnata.*

La prima operazione da fare è rimuovere la finestra, nel caso in cui questa sia visibile a video, utilizzando le stesse istruzioni impiegate per *WindowHide()*.

```
DECLARE SUB WindowDelete (Handle%)

   CurRow% = CSRLIN
   CurCol% = POS(0)

   DIM InRegs AS RegType, OutRegs AS RegType

   REM Se necessario ripristina il vecchio testo.

   IF OnFlag(Handle%) THEN

      TR = TopRow(Handle%)
      TC = TopCol(Handle%)
      LOCATE TR, TC
      InRegs.cx = 1

      FOR i = 1 TO Rows(Handle%)
         FOR j = 1 TO Cols(Handle%)
            InRegs.ax = &H900 + ASC(MID$(OldText(Handle%, i), j, 1))
            InRegs.bx = ASC(MID$(OldAttrb(Handle%, i), j, 1))
            CALL INTERRUPT(&H10, InRegs, OutRegs)
            LOCATE TR + i - 1, TC + j
         NEXT j
            LOCATE TR + i, TC
      NEXT i

   END IF

   LOCATE CurRow%, CurCol%
   :
   :
END SUB
```

Si è pronti adesso ad avviare la reale operatività di *WindowDelete( )* che dovrà comunicare a *WindowGetHandle%( )* che l'identificatore della finestra è vuoto. La funzione *WindowGetHandle%( )* controlla le voci che risultano vuote negli array delle finestre controllando l'array *Rows( )* come di seguito riportato:

```
FOR i = 1 TO MaxWindows
   IF Rows(i) = 0 THEN
      WindowGetHandle% = 1
      TopRow(i) = TopR%
      TopCol(i) = TopC%
      BotRow(i) = TopR% + NRow%
      BotCol(i) = TopC% + NCol%
      Rows(i) = NRow%
      Cols(i) = NCol%
      Attribute(i) = Attr%
      OnFlag(i) = 0
      FOR j = 1 TO Rows(i)
         Text(i, j) = SPACE$(Cols(i))
      NEXT j
   EXIT FOR
   END IF
NEXT i
```

In altre parole, per cancellare una finestra sarà sufficiente impostare *Rows(Handle%)* a 0, dove *Handle%* corrisponde all'identificatore che è stato trasferito. Questa è un'operazione molto semplice che richiede l'inserimento di una sola riga. Il listato 2.6 presenta l'intero sottoprogramma.

```
DECLARE SUB WindowDelete (Handle%)

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)
    CONST MaxWindows = 8
    DIM Rows(1 TO MaxWindows) AS INTEGER
```

continua

Listato 2.6 *Sottoprogramma WindowDelete per la cancellazione delle finestre.*

```
    DIM Cols(1 TO MaxWindows) AS INTEGER
    DIM TopRow(1 TO MaxWindows) AS INTEGER
    DIM TopCol(1 TO MaxWindows) AS INTEGER
    DIM BotRow(1 TO MaxWindows) AS INTEGER
    DIM BotCol(1 TO MaxWindows) AS INTEGER
    DIM Attribute(1 TO MaxWindows) AS INTEGER
    DIM OnFlag(1 TO MaxWindows) AS INTEGER
    DIM Text(MaxWindows, 25) AS STRING
    DIM OldText(MaxWindows, 25) AS STRING
    DIM OldAttrb(MaxWindows, 25) AS STRING
    COMMON SHARED /WindowA/ Rows() AS INTEGER, Cols() AS INTEGER,
         TopRow() AS INTEGER, TopCol() AS INTEGER,
         BotRow() AS INTEGER, BotCol() AS INTEGER
     COMMON SHARED /WindowB/ Attribute() AS INTEGER,
         OnFlag() AS INTEGER, Text() AS STRING,
         OldText() AS STRING, OldAttrb() AS STRING

SUB WindowDelete (Handle%)

    CurRow% = CSRLIN
    CurCol% = POS(0)

    DIM InRegs AS RegType, OutRegs AS RegType
    REM Se necessario ripristina il vecchio testo.

    IF OnFlag(Handle%) THEN

       TR = TopRow(Handle%)
       TC = TopCol(Handle%)
       LOCATE TR, TC
       InRegs.cx = 1
       FOR i = 1 TO Rows(Handle%)
          FOR j = 1 TO Cols(Handle%)
             InRegs.ax = &H900 + ASC(MID$(OldText(Handle%, i),
                j, 1))
             InRegs.bx = ASC(MID$(OldAttrb(Handle%, i), j, 1))
             CALL INTERRUPT(&H10, InRegs, OutRegs)
             LOCATE TR + i - 1, TC + j
          NEXT j
             LOCATE TR + i, TC
       NEXT i
    END IF

    LOCATE CurRow%, CurCol%
    Rows(Handle%) = 0    'WindowGetHandle controlla solo questo.
END SUB
```

Listato 2.6 *Sottoprogramma WindowDelete per la cancellazione delle finestre.*

E con questo si è anche visto come creare un sottoprogramma per la cancellazione delle finestre avendo così a disposizione uno strumento che consente di eseguire alcune, a volte necessarie, operazioni di pulizia. Si utilizzi il sottoprogramma *WindowDelete( )* ogni volta che si sta raggiungendo il numero massimo di finestre che il sistema consente di memorizzare per liberare uno o più gestori.

PROVA DI CANCELLAZIONE DI UNA FINESTRA

Di seguito viene mostrato come avviare *WindowDelete( )*:

```
DECLARE FUNCTION WindowGetHandle% (TopRow%, TopCol%, NumRow%,
                                   NumCol%, Attr%)
DECLARE SUB WindowShow (Handle%)
DECLARE SUB WindowDelete (Handle%)

Handle1% = WindowGetHandle%(10, 10, 5, 5, &H24)
Handle2% = WindowGetHandle%(10, 10, 3, 13, &H61)

CALL WindowShow(Handle1%)
CALL WindowShow(Handle2%)
CALL WindowDelete(Handle2%)
```

Questo programma visualizza una finestra e la copre immediatamente con un'altra. Viene a questo punto cancellata la seconda finestra e il relativo gestore viene liberato per poter essere successivamente riassegnato.

Con questo si è terminata la trattazione di un sistema basato su finestre. Si sono progettate, visualizzate, nascoste, compilate, spostate e cancellate le finestre. Si è così venuti a conoscenza di un valido strumento di programmazione ma bisogna anche tenere presente che il BASIC *PDS* contiene altri strumenti per la gestione delle finestre che verranno descritti nel successivo paragrafo di questo libro.

IL BASIC PDS E LE FINESTRE

Il *BASIC Professional Development System* (PDS) contiene alcuni strumenti per la gestione delle finestre attivabili direttamente. Per vedere come funzionano questi strumenti aggiuntivi, si proverà ora a generare una finestra contenente la parola "Salve!" lampeggiante a video.

La Microsoft fornisce alcune routine in linguaggio assembly già predisposte per una rapida operazione di presentazione e scomparsa delle finestre e questo codice è contenuto in un file denominato *UIASM.OBJ* (fornito con il BASIC PDS). Se si sta utilizzando *QBX.EXE* sarà necessario immettere questo file .OBJ in una particolare

libreria *Quick* prima di poterlo utilizzare. Se si sta utilizzando BC.EXE (versione 7.0 o successiva), questo file .OBJ potrà essere immesso in un file libreria per una successiva operazione di linking. Avviare quindi QBX.EXE, dopo aver creato la libreria *UIASM.QLB* come di seguito riportato:

```
LINK /Q UIASM.OBJ QBX.LIB, UIASM.QLB,, QBXQLB.LIB;
```

> ***Consiglio:*** È buona norma creare una propria libreria Quick personalizzata alla quale avviare i moduli compilati poiché questi verranno caricati tutti assieme rendendo più veloce l'operazione di compilazione di un programma.

Se, per esempio, si volesse richiamare un programma a cui è stato assegnato il nome WINDOWER.BAS, questo dovrebbe venire caricato in QBX.EXE come di seguito riportato:

```
QBX WINDOWER /L UIASM
```

Questa riga di comando carica WINDOWER.BAS e la libreria Quick precedentemente creata. Prima di poter avviare il programma sarà necessario caricare i file PDS *MENU.BAS*, *GENERAL.BAS*, *WINDOW.BAS* e *MOUSE.BAS* poiché il sistema a finestre del BASIC PDS utilizza codici provenienti da questi moduli. A questo punto si è pronti a iniziare e WINDOWER potrà essere avviato per fare apparire le finestre.
Se si sta utilizzando BC.EXE, si potrà creare una libreria a cui assegnare il nome UIASM.LIB in grado di contenere tutti i moduli che saranno necessari. Questo file dovrà contenere non solo il modulo UIASM.OBJ ma anche le versioni compilate di MENU.BAS, GENERAL.BAS, WINDOW.BAS e MOUSE.BAS. Si compilino questi file con *BC.EXE* e si crei una libreria UIASM.LIB come di seguito riportato:

```
LIB UIASM.LIB+UIASM.OBJ+GENERAL.OBJ+MOUSE.OBJ+MENU.OBJ
+WINDOW.OBJ+QBX.LIB;
```

Adesso che si ha tutto a disposizione, compilare WINDOWER.BAS con BC.EXE e in seguito creare il file .EXE come di seguito riportato:

```
LINK WINDOWER,,,UIASM
```

A questo punto sarà possibile avviare WINDOWER.EXE e ottenere lo stesso risultato generato da QBX.EXE.

WINDOWER.BAS: UN PROGRAMMA A FINESTRE DI BASIC PDS

Si vedrà ora il programma WINDOWER.BAS che genera l'immissione a video di una finestra. Il programma dovrà essere iniziato con una serie di definizioni, come di seguito riportato:

```
' $INCLUDE: 'C:\BC7\WINDOW.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\GENERAL.BI'
  :
```

Il passo successivo consiste nell'impostare alcune istruzioni *COMMON*:

```
' $INCLUDE: 'C:\BC7\WINDOW.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\GENERAL.BI'

COMMON SHARED /uitools/ GloMenu            AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()         AS MenuTitleType
COMMON SHARED /uitools/ GloItem()          AS MenuItemType
COMMON SHARED /uitools/ GloWindow()        AS windowType
COMMON SHARED /uitools/ GloButton()        AS buttonType
COMMON SHARED /uitools/ GloEdit()          AS EditFieldType
COMMON SHARED /uitools/ GloStorage         AS WindowStorageType
COMMON SHARED /uitools/ GloWindowStack()   AS INTEGER
COMMON SHARED /uitools/ GloBuffer$()

DIM GloTitle(MAXMENU)            AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)    AS MenuItemType
DIM GloWindow(MAXWINDOW)         AS windowType
DIM GloButton(MAXBUTTON)         AS buttonType
DIM GloEdit(MAXEDITFIELD)        AS EditFieldType
DIM GloWindowStack(MAXWINDOW)    AS INTEGER
DIM GloBuffer$(MAXWINDOW + 1, 2)
:
:
```

Adesso si potrà avviare il processo di inizializzazione. Per visualizzare sullo schermo una finestra sarà necessario inizializzare tre sistemi: il sistema a finestre, il sistema mouse e il sistema menu (anche se non si prevede di utilizzare né i menu né il mouse).

```
' $INCLUDE: 'C:\BC7\WINDOW.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\GENERAL.BI'
COMMON SHARED /uitools/ GloMenu            AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()         AS MenuTitleType
COMMON SHARED /uitools/ GloItem()          AS MenuItemType
COMMON SHARED /uitools/ GloWindow()        AS windowType
COMMON SHARED /uitools/ GloButton()        AS buttonType
COMMON SHARED /uitools/ GloEdit()          AS EditFieldType
COMMON SHARED /uitools/ GloStorage         AS WindowStorageType
COMMON SHARED /uitools/ GloWindowStack()   AS INTEGER
COMMON SHARED /uitools/ GloBuffer$()
```

```
DIM GloTitle(MAXMENU)              AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)      AS MenuItemType
DIM GloWindow(MAXWINDOW)           AS windowType
DIM GloButton(MAXBUTTON)           AS buttonType
DIM GloEdit(MAXEDITFIELD)          AS EditFieldType
DIM GloWindowStack(MAXWINDOW)      AS INTEGER
DIM GloBuffer$(MAXWINDOW + 1, 2)
   CALL MenuInit
   CALL WindowInit
   CALL MouseShow
   :
```

A questo punto si è quasi pronti alla visualizzazione della finestra tramite il sottoprogramma PDS *WindowOpen( )*. Prima di tutto, sarà necessario immettere i valori di numerosi argomenti che dovranno essere trasferiti al sottoprogramma, come di seguito riportato:

Nome parametro	Descrizione
Handle%	Numero della finestra (per esempio: 1)
Row1%, Col1%	Coordinate dell'angolo superiore sinistro
Row2%, Col2%	Coordinate dell'angolo inferiore destro
TextFore%	Colore di primo piano del testo (0-15)
TextBack%	Colore dello sfondo del testo (0-7)
Fore%	Colore primo piano finestra (0-15)
Back%	Colore sfondo finestra (0-7)
Highlight%	Colore dei pulsanti in alta luminosità (0-15)
Movewin%	Se *TRUE* (definito dai file .BI) la finestra potrà essere spostata
Closewin%	Se *TRUE* la finestra potrà essere chiusa
Sizewin%	Se *TRUE*, la finestra potrà essere ridimensionata
Modalwin%	*TRUE* significa che una selezione all'esterno della finestra genera un segnale sonoro
Borderchar%	0 = Nessun bordo 1 = Bordo semplice 2 = Bordo doppio
Title$	Titolo della finestra: "" = nessun titolo

Vengono di seguito riportati i valori impostati per la finestra di esempio:

```
REM E' necessario Microsoft BASIC PDS.

' $INCLUDE: 'C:\BC7\WINDOW.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
```

```
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\GENERAL.BI'

COMMON SHARED /uitools/ GloMenu            AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()         AS MenuTitleType
COMMON SHARED /uitools/ GloItem()          AS MenuItemType
COMMON SHARED /uitools/ GloWindow()        AS windowType
COMMON SHARED /uitools/ GloButton()        AS buttonType
COMMON SHARED /uitools/ GloEdit()          AS EditFieldType
COMMON SHARED /uitools/ GloStorage         AS WindowStorageType
COMMON SHARED /uitools/ GloWindowStack()   AS INTEGER
COMMON SHARED /uitools/ GloBuffer$()

DIM GloTitle(MAXMENU)            AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)    AS MenuItemType
DIM GloWindow(MAXWINDOW)         AS windowType
DIM GloButton(MAXBUTTON)         AS buttonType
DIM GloEdit(MAXEDITFIELD)        AS EditFieldType
DIM GloWindowStack(MAXWINDOW)    AS INTEGER
DIM GloBuffer$(MAXWINDOW + 1, 2)
   CALL MenuInit
   CALL WindowInit
   CALL MouseShow

   CALL WindowOpen(1, 10, 20, 15, 30, 2, 1, 2, 1, 15,
      TRUE, TRUE, TRUE, TRUE, 0, "")
```

Dopo aver completato quanto sopra, la finestra dovrebbe venire visualizzata sullo schermo non prima, però, di aver posizionato il cursore con *WindowLocate( )*. A questo punto si potrà utilizzare il sottoprogramma PDS *WindowPrint( )* per generare la visualizzazione della finestra (listato 2.7).

```
REM E' necessario Microsoft BASIC PDS.

' $INCLUDE: 'C:\BC7\WINDOW.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\GENERAL.BI'

COMMON SHARED /uitools/ GloMenu            AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()         AS MenuTitleType
COMMON SHARED /uitools/ GloItem()          AS MenuItemType
COMMON SHARED /uitools/ GloWindow()        AS windowType
COMMON SHARED /uitools/ GloButton()        AS buttonType
COMMON SHARED /uitools/ GloEdit()          AS EditFieldType
```

continua

Listato 2.7 *Uso delle finestre con BASIC PDS.*

```
COMMON SHARED /uitools/ GloStorage          AS WindowStorageType
COMMON SHARED /uitools/ GloWindowStack()    AS INTEGER
COMMON SHARED /uitools/ GloBuffer$()

DIM GloTitle(MAXMENU)              AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)      AS MenuItemType
DIM GloWindow(MAXWINDOW)           AS windowType
DIM GloButton(MAXBUTTON)           AS buttonType
DIM GloEdit(MAXEDITFIELD)          AS EditFieldType
DIM GloWindowStack(MAXWINDOW)      AS INTEGER
DIM GloBuffer$(MAXWINDOW + 1, 2)

    CALL MenuInit
    CALL WindowInit
    CALL MouseShow

    CALL WindowOpen(1, 10, 20, 15, 30, 2, 1, 2, 1, 15, TRUE,
       TRUE, TRUE, TRUE, 0, "")
    CALL WindowLocate(1, 1)
    CALL WindowPrint(1, "Salve!")
```

Listato 2.7 *Uso delle finestre con BASIC PDS.*

In questo esempio, è stata utilizzata *WindowLocate( )* per posizionare il cursore a (1, 1) all'interno della finestra e in seguito *WindowPrint( )* per visualizzare la finestra 1. A questo punto nella finestra dovrebbe apparire la parola "Salve!" e il processo è terminato. Se si possiede BASIC PDS si provi a utilizzarlo poiché esso contiene molte altre funzioni per le finestre e sicuramente alcune di queste potrebbero adattarsi alle specifiche esigenze di ciascun utente.

CONCLUSIONE

In questo capitolo si è analizzata la procedura per impostare, visualizzare o rimuovere, spostare e cancellare le finestre e si è anche visto che il BASIC PDS consente una gestione più semplice delle finestre. Nel prossimo capitolo, verrà analizzato l'uso del mouse.

CAPITOLO 3

MOUSE

In questo capitolo verrà presa in esame la gestione del mouse servendosi dell'interrupt DOS &H33 (interrupt del mouse) in modo da poter gestire correttamente il mouse anche con il BASIC attivando spesso la routine *INTERRUPT( )*.

***Nota**: Poiché i sottoprogrammi e le funzioni di questo capitolo utilizzano la routine INTERRUPT( ), sarà necessario che questa venga caricata, se si sta utilizzando QuickBasic (QB.EXE oppure QBX.EXE) con l'opzione /L.*

Si vedrà qui come inizializzare il mouse, come leggerne la posizione, come reperire le informazioni trasmesse attraverso la pressione dei pulsanti e, infine, come operare con il puntatore. Verranno inoltre presentati esempi che spiegano come utilizzare le informazioni del mouse e, nel capitolo successivo, si metterà in pratica quanto appreso qui appreso quando verrà richiesto di associare il mouse alla generazione dei menu a tendina.

PRIMA DI INIZIARE

Prima di iniziare a operare con il mouse è necessario prendere in considerazione due concetti basilari. Innanzitutto, in ambiente DOS, prima di poter utilizzare un mouse si dovrà caricare in memoria il software di gestione del mouse avviando il relativo file *.COM* che viene fornito con il mouse stesso (per esempio, *MOUSE.COM* per un mouse della Microsoft o della Logitech, oppure *MOUSESYS.COM* per un mouse del tipo *Mouse System*). Questo programma di controllo dovrà essere avviato prima di qualsiasi altro programma che consente l'utilizzo del mouse: per ulteriori

informazioni, si consiglia di prendere visione della documentazione che accompagna il mouse stesso. Si noti che in ambiente OS/2, oppure in emulazione DOS di OS/2 non sarà necessario caricare questo programma di controllo in quanto esso viene automaticamente attivato.

***Nota**: Il programma di controllo del mouse caricherà in memoria il codice di gestione dell'interrupt del mouse, corrispondente a &H33.*

In secondo luogo, prima di poterlo utilizzare, il mouse dovrà essere inizializzato: proprio come successo per le finestre di cui si è trattato nel precedente capitolo, dove si è utilizzato *WindowGetHandle%( )*, anche il mouse richiede un'operazione di inizializzazione. Questo capitolo inizierà proprio con la creazione di una funzione a cui è stato assegnato il nome *MouseInitialize%( )* il cui scopo è esattamente quello di inizializzare questo tipo di periferica. Questa funzione dovrà quindi essere chiamata ogni volta che si desidera utilizzare il mouse e solo successivamente potranno essere utilizzate altre funzioni mouse o altri sottoprogrammi che ne prevedono l'uso.

***Nota**: La funzione MouseInitialize%( ) inizializza il software di gestione del mouse precedentemente caricato in memoria.*

***Consiglio**:* In molti programmi, l'uso del mouse è facoltativo e dipende dall'installazione o meno di un mouse e del relativo software di gestione. È necessario sapere, a questo punto, che, anche se l'uso del mouse è facoltativo, potranno sempre essere richiamate senza alcun problema funzioni e sottoprogrammi che ne prevedono l'uso. Se il mouse non è stato installato verranno semplicemente disabilitate operazioni quali lo spostamento del cursore o la pressione dei tasti senza che venga generato un loop perpetuo in attesa di queste operazioni. In altre parole, l'uso dei sottoprogrammi e funzioni che prevedono il mouse non generano alcun problema anche se il mouse non è fisicamente presente.

INIZIALIZZAZIONE DEL MOUSE

L'inizializzazione del mouse è la prima operazione da avviare ed essa prevede l'uso dell'interrupt &H33, servizio 0. Questo servizio restituisce un valore diverso da zero nel caso in cui sia stato inizializzato il mouse altrimenti il mouse non può essere utilizzato, (mouse non installato o software di gestione inesistente); in questo caso, e se il programma non può funzionare senza mouse, si dovrà stampare un messaggio di errore e uscire dalla routine.

Quando il mouse è stato inizializzato, questo rimarrà attivo fintanto che il computer non verrà spento, perciò non sarà necessario inizializzarlo nuovamente (a meno che non siano sorti problemi di gestione) ad ogni funzione e/o sottoprogramma. Si noti che l'inizializzazione del mouse non visualizza immediatamente il *puntatore* poiché la sua presenza dovrà essere definita da un particolare sottoprogramma che verrà analizzato in seguito e al quale è stato assegnato il nome *MouseShowCursor( )*.

LA FUNZIONE MouseInitialize%( )

Questa funzione, per poter inizializzare il sistema mouse, utilizza l'interrupt &H33, servizio 0 (ossia: ah=0). Come già detto in precedenza, questo è il primo passo necessario da intraprendere prima di poter avviare qualsiasi funzione a mouse. Si carichi per cui *ah* con zero, si richiami l'interrupt &H33 e in seguito si imposti *MouseInitialize%* al valore che viene restituito da *ah*.

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.ax = 0
CALL INTERRUPT (&H33, InRegs, OutRegs)
MouseInitialize% = OutRegs.ax
```

Se il valore trasferito è diverso da zero, il mouse verrà inizializzato, in caso contrario l'operazione verrà interrotta. Successivamente bisogna controllare il valore restituito. Il listato 3.1 mostra l'intera funzione di inizializzazione del mouse.

```
DECLARE FUNCTION MouseInitialize% ()
TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)
FUNCTION MouseInitialize%
    DIM InRegs AS RegType, OutRegs AS RegType
    InRegs.ax = 0
    CALL INTERRUPT(&H33, InRegs, OutRegs)
    MouseInitialize% = OutRegs.ax
END FUNCTION
```

Listato 3.1 *La funzione MouseInitialize%( ).*

ESEMPIO DI INIZIALIZZAZIONE DEL MOUSE

Si vedrà ora come utilizzare la funzione *MouseInitialize%( )*.

```
REM Esempio di inizializzazione del mouse

DECLARE FUNCTION MouseInitialize% ( )

LOCATE 1,1

IF MouseInitialize% THEN
   PRINT "Mouse inizializzato."
ELSE
   PRINT "Driver del mouse non installato."
END IF
```

Qui viene controllato il valore restituito da *MouseInitialize%( )* e, nel caso in cui questo sia pari a zero verrà emesso il messaggio di errore "Driver del mouse non installato." e la procedura verrà interrotta; in caso contrario, viene emesso il messaggio "Mouse inizializzato.".

VISUALIZZAZIONE DEL PUNTATORE DEL MOUSE

Il passo successivo è quello legato alla visualizzazione del *puntatore* del mouse. In modalità testo, il puntatore assumerà l'aspetto di un rettangolo luminoso (anche se esso potrà essere modificato con *MouseSetCursor( )* di cui si tratterà più avanti in questo capitolo) e, in modalità grafica, assumerà l'aspetto di una freccia. Verrà ora presentato un breve sottoprogramma, a cui verrà assegnato il nome *MouseShowCursor( )*, il cui scopo è quello di visualizzare il puntatore del mouse.

IL SOTTOPROGRAMMA MouseShowCursor( )

Come qualsiasi altra funzione e/o sottoprogramma per il mouse, verrà anche in questa occasione utilizzato l'interrupt &H33, ma, in questo caso, prendendo in considerazione il servizio 1, che visualizza il puntatore del mouse.

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.ax = 1
CALL INTERRUPT(&H33, InRegs, OutRegs)
```

Quando verrà utilizzato *MouseShowCursor( )*, a video verrà riportato il puntatore del mouse. Il listato 3.2 presenta il sottoprogramma completo.

```
DECLARE SUB MouseShowCursor ()

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

SUB MouseShowCursor

    DIM InRegs AS RegType, OutRegs AS RegType
    InRegs.ax = 1
    CALL INTERRUPT(&H33, InRegs, OutRegs)

END SUB
```

Listato 3.2 *Il sottoprogramma MouseShowCursor( ).*

USO DI MouseShowCursor( )

Di seguito viene indicato come utilizzare il sottoprogramma *MouseShowCursor( )*.

```
REM Esempio di MouseShowCursor

DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )

Check% = MouseInitialize%

LOCATE 1,1

IF Check% = 0 THEN
   PRINT "Driver del mouse non installato."
ELSE
```

```
    PRINT "Mouse inizializzato."
    CALL MouseShowCursor
  END IF
```

Si noti che nell'esempio sopra riportato prima è stato inizializzato il mouse con la funzione *MouseInitialize%( )* e, se l'inizializzazione è stata correttamente effettuata, si è richiesta la visualizzazione del puntatore invocando il sottoprogramma *MouseShowCursor( )*.

> ***Consiglio***: Se per una qualsiasi ragione non è stato inizializzato il mouse, le chiamate all'interrupt &H33 non avranno alcun effetto.

Ora che è stato visualizzato il puntatore del mouse, si vedrà come nasconderne la presenza.

COME NASCONDERE IL PUNTATORE DEL MOUSE

Vi sono occasioni in cui il puntatore del mouse dovrebbe venire nascosto: questo paragrafo indica come eseguire questa operazione.

Nota: *Nel caso in cui il puntatore del mouse sia già invisibile, esso rimarrà nascosto anche quando viene avviata la procedura per nasconderlo.*

Vi è una semplice — ma a volte molto importante — ragione per nascondere il puntatore del mouse: quando il puntatore del mouse viene spostato all'interno dello schermo, il software di gestione del mouse legge continuamente i caratteri che vengono "toccati" dal mouse prima di visualizzare la reale posizione del puntatore. In seguito, quando il puntatore viene spostato, il carattere su cui esso si trovava viene ripristinato, ivi compresi gli attributi ad esso assegnati. Tuttavia, se la visualizzazione di quanto sta sotto il puntatore viene modificata mentre il mouse ricopre un carattere, verrà successivamente ripristinato l'attributo del carattere (scorretto) che stava inizialmente sotto il puntatore.
Per esempio, se il puntatore del mouse viene ipoteticamente spostato alla posizione *A*, il driver leggerà il carattere a tale posizione per poterlo successivamente ripristinare e, poi, lo sovrascrive con il puntatore del mouse. Ora, se prima di spostare il puntatore, viene attivata, con *WindowShow( )*, una finestra e successivamente si sposta il puntatore, il software di gestione del mouse, indipendentemente dall'aspet-

to della finestra, sostituirà il carattere originale nella posizione *A*, generando un "buco" all'interno della finestra.
Per evitare l'insorgere di questo problema, sarà sempre necessario *disattivare* il puntatore del mouse quando si visualizza una finestra o quando si desidera sovrascrivere la posizione del puntatore del mouse con un sottoprogramma, a cui si assegna il nome *MouseHideCursor( )* e riattivare immediatamente dopo la visualizzazione (avviando nuovamente il sottoprogramma *MouseShowCursor( )*). L'intera procedura risolve completamente i problemi di sovrascrittura.

***Nota**: In effetti, se si analizzano attentamente i programmi per la creazione dei menu presentati nel capitolo 4, si noterà dove e come operare per evitare problemi come quelli sopra specificati.*

Ora verrà realizzato un sottoprogramma per nascondere il mouse a cui verrà assegnato il nome *MouseHideCursor( )*.

IL SOTTOPROGRAMMA MouseHideCursor( )

Per la realizzazione di questo sottoprogramma si farà uso dell'interrupt &H33, servizio 2 che nasconde la presenza del puntatore del mouse.

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.ax = 2
CALL INTERRUPT(&H33, InRegs, OutRegs)
```

Il listato 3.3 presenta l'intero sottoprogramma.

```
DECLARE SUB MouseHideCursor ()

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)
```

continua

Listato 3.3 *Il sottoprogramma MouseHideCursor( )*

```
SUB MouseHideCursor

    DIM InRegs AS RegType, OutRegs AS RegType
    InRegs.ax = 2
    CALL INTERRUPT(&H33, InRegs, OutRegs)

END SUB
```

Listato 3.3 *Il sottoprogramma MouseHideCursor( ) (continua)*

USO DEL SOTTOPROGRAMMA MouseHideCursor( )

L'esempio riportato nel listato 3.4 visualizza il puntatore del mouse e in seguito ne nasconde la presenza dopo aver premuto un qualsiasi tasto.

```
REM Esempio di MouseHideCursor

DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE SUB MouseHideCursor ( )
Check% = MouseInitialize%

LOCATE 1,1
IF Check% = 0 THEN
    PRINT "Driver del mouse non installato."
ELSE
    PRINT "Mouse inizializzato."
    CALL MouseShowCursor
    PRINT "Premere un tasto per nascondere il puntatore."
    DO
    LOOP WHILE INKEY$ = " "
    CALL MouseHideCursor
END IF
```

Listato 3.4 *Il programma per nascondere il puntatore del mouse.*

Ancora una volta, è stato prima inizializzato il sistema mouse e, se l'operazione è stata correttamente eseguita, viene visualizzato il puntatore ed emesso il messaggio "Premere un tasto per nascondere il puntatore.". Dopo la pressione del tasto il puntatore scomparirà dal video grazie all'invocazione del sottoprogramma *MouseHideCursor( )*.

A questo punto si è terminata l'impostazione del sistema mouse e la visualizzazione o meno del suo puntatore. Ora si passerà, invece, all'interpretazione delle informazioni lette dal mouse.
Per esempio, vi sono occasioni in cui si desidera controllare lo stato del mouse, quale pulsante è stato premuto o dove si trova il puntatore. Il prossimo paragrafo spiega come effettuare queste operazioni.

LETTURA IMMEDIATA DELLE INFORMAZIONI DEL MOUSE

Vi è un solo sistema per leggere le informazioni generate dal mouse: usando l'interrupt &H33, servizio 3, che restituisce immediatamente lo stato dei pulsanti destro e sinistro del mouse e la riga e la colonna in cui è posizionato il suo puntatore. Quando viene richiamato questo interrupt, verranno restituite le informazioni codificate nei registri *bx, cx* e *dx*, che forniscono lo stato attuale del mouse.
Di seguito vengono riportati i valori restituiti dei registri.

Registri	*Valori restituiti*
bx	0 = nessun pulsante premuto; 1 = pressione del pulsante destro; 2 = pressione del pulsante sinistro; 3 = pressione di entrambi i pulsanti.
cx	Posizione di colonna del puntatore del mouse (calcolato in pixel).
dx	Riga su cui è posizionato il puntatore del mouse (espressa in pixel).

Verrà ora realizzato un sottoprogramma che restituisca queste informazioni.

IL SOTTOPROGRAMMA MouseInformation( )

L'ideale è generare un sottoprogramma che potrà essere definito come segue: *CALL MouseInformation (Right%, Left%, Row%, Col%).* Le variabili che verranno trasferite potranno restituire i valori di seguito specificati:

Variabili	*Valori restituiti*
Right%	0 = Pulsante destro non premuto; 1 = Pulsante destro premuto.
Left%	0 = Pulsante sinistro non premuto; 1 = Pulsante sinistro premuto.
Row%	Riga sulla quale è posizionato il cursore in modalità testo (da 1 a 25).
Col%	Colonna sulla quale è posizionato il cursore in modalità testo (da 1 a 80).

Lo scopo di questo sottoprogramma è quello di restituire una serie di informazioni sullo stato del mouse. Come si è già visto, l'interrupt &H33, servizio 3 restituisce le informazioni sui pulsanti del mouse in *OutRegs.bx*: se questo valore è pari a 1, significa che è stato premuto il pulsante sinistro; se il valore è pari a 2 significa che è stato premuto il pulsante destro; se è pari a 3 significa che sono stati premuti entrambi i pulsanti del mouse. La restituzione di 0 significa, invece, che non è stato premuto alcun pulsante. Si vedrà ora come decodificare le informazioni dei parametri *Right%* e *Left%*.

```
SUB MouseInformation (Right%, Left%, Row%, Col%)
   DIM InRegs AS RegType, OutRegs AS RegType
   InRegs.ax = 3
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   Right% = 0
   Left% = 0

   SELECT CASE OutRegs.bx
      CASE 1
         Left% = 1
      CASE 2
         Right% = 1
      CASE 3
         Left% = 1
         Right% = 1
   END SELECT
   :
   :
```

OutRegs.dx contiene il numero di riga (espresso in pixel) e *OutRegs.cx* contiene il numero di colonna (anch'esso espresso in pixel). Il lavoro che viene fatto sul mouse e sui menu dovrà essere eseguito in modalità testo poiché questa consente di utilizzare il valore reale delle colonne e delle righe video (ossia: 1-25 per le righe e 1-80 per le colonne) e non dei pixel (che, rispettivamente, sono 0-199 e 0-639). Per convertire i valori da pixel in righe e colonne, si utilizzi l'esempio di seguito riportato:

```
SUB MouseInformation (Right%, Left%, Row%, Col%)

   DIM InRegs AS RegType, OutRegs AS RegType
   InRegs.ax = 3
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   Right% = 0
   Left% = 0

   SELECT CASE OutRegs.bx
      CASE 1
         Left% = 1
```

```
      CASE 2
         Right% = 1
      CASE 3
         Left% = 1
         Right% = 1
   END SELECT

   Row% = OutRegs.dx \ 8 + 1
   Col% = OutRegs.cx \ 8 + 1
```

E con questo si è terminato. Il listato 3.5 presenta l'intero sottoprogramma.

```
DECLARE SUB MouseInformation (Right%, Left%, Row%, Col%)

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

SUB MouseInformation (Right%, Left%, Row%, Col%)

    DIM InRegs AS RegType, OutRegs AS RegType
    InRegs.ax = 3
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    Right% = 0
    Left% = 0

    SELECT CASE OutRegs.bx
       CASE 1
          Left% = 1
       CASE 2
          Right% = 1
       CASE 3
          Left% = 1
          Right% = 1
    END SELECT
```

continua

Listato 3.5 *Il sottoprogramma MouseInformation( ).*

```
    Row% = OutRegs.dx \ 8 + 1
    Col% = OutRegs.cx \ 8 + 1

END SUB
```

Listato 3.5 *Il sottoprogramma MouseInformation( )*

CONTROLLO DELLO STATO DEL MOUSE

Il programma di seguito riportato presenta lo stato del mouse quando viene premuto un qualsiasi tasto.

```
REM Esempio di MouseInformation

DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE SUB MouseHideCursor ( )
Check% = MouseInitialize%

LOCATE 1,1
IF Check% = 0 THEN
   PRINT "Driver del mouse non installato."
ELSE
   PRINT "Mouse inizializzato."
   CALL MouseShowCursor
   PRINT "Posizionare il puntatore e premere un tasto."
   DO
   LOOP WHILE INKEY$ = " "
   CALL MouseInformation (Right%, Left%, Row%, Col%)

   IF Right% THEN
      PRINT "Pulsante destro premuto."
   ELSE
      PRINT "Pulsante destro non premuto."
   END IF

   IF Left% THEN
      PRINT "Pulsante sinistro premuto."
   ELSE
      PRINT "Pulsante sinistro non premuto."
   END IF

   PRINT "Riga: ", Row%, "Colonna: ", Col%

END IF
```

L'esempio precedentemente riportato è un possibile uso del sottoprogramma *MouseInformation( )* e della successiva decodifica dei valori restituiti da *Right%*, *Left%*, *Row%* e *Col%*. Per poter utilizzare questo programma bisogna accertarsi di aver precedentemente caricato il software di gestione del mouse, come descritto all'inizio di questo capitolo. Quando verrà premuto un qualsiasi tasto il programma restituirà un rapporto sullo stato attuale del mouse.
Una delle più grosse limitazioni del sottoprogramma *MouseInformation( )* è data dal fatto che vengono restituiti i dati relativi solamente alla condizione attuale del mouse e se lo si desidera utilizzare per le immissioni eseguite con il mouse stesso sarà necessario generare un loop che continuerà fino a che non succederà qualcosa.

***Consiglio**:* Una soluzione migliore è data dai sottoprogrammi che verranno successivamente presentati in questo capitolo e che faranno uso di servizi specializzati dell'interrupt &H33. In questi programmi, per esempio, le varie pressioni dei tasti verranno "messe in coda" fino a quando non ne verrà richiesto un rapporto. La procedura della "coda" è una pratica che viene spesso utilizzata in fase di programmazione poiché non viene richiesta l'emissione di un rapporto ad ogni pressione di un tasto, ma questa potrà essere generata solo quando è realmente necessaria.

Inoltre, vi saranno certamente occasioni in cui si desidera che i valori di riga e di colonna vengano restituiti con valori espressi in pixel (invece che in valori di riga e colonna). In questo caso, sarà sufficiente impostare, alla fine di *MouseInformation( )*, *Row%* e *Col%* rispettivamente uguali a *OutRegs.dx* e *OutRegs.cx*.

```
Row% = OutRegs.dx
Col% = OutRegs.cx
```

***Consiglio**:* In modalità testo il puntatore del mouse salta da una posizione di carattere a una successiva e il pixel che viene restituito corrisponde all'angolo superiore sinistro della posizione del carattere.

Si passerà ora all'esplorazione del servizio 4 dell'interrupt &H33 che consente di spostare il puntatore del mouse in una qualsiasi posizione del video.

SPOSTAMENTO DEL PUNTATORE DEL MOUSE

Il servizio 4 dell'interrupt &H33 consente di controllare gli spostamenti del puntatore del mouse. Fino a ora, gli spostamenti del puntatore sono stati strettamente collegati allo spostamento del mouse stesso (in effetti, ogni spostamento del mouse genera uno spostamento del puntatore a video), tuttavia, è sempre possibile realizzare un sottoprogramma, a cui assegnare il nome *MouseMoveCursor( )*, in grado di definire un'esatta posizione del puntatore del mouse.

IL SOTTOPROGRAMMA MouseMoveCursor( )

Indubbiamente, il sistema più semplice per spostare il puntatore è quello di trasferirgli le coordinate di riga e colonna come di seguito riportato:

```
CALL MouseMoveCursor (Row%, Col%)
```

Per scrivere il sottoprogramma *MouseMoveCursor( )* si dovrà utilizzare l'interrupt &H33, servizio 4. Sarà necessario trasferire la nuova posizione del puntatore ai registri *dx* (riga) e *cx* (colonna) e, poiché si stanno utilizzando le coordinate in modalità testo di riga e colonna, sarà prima necessario convertire i valori in pixel come di seguito specificato:

```
DIM InRegs AS RegType, OutRegs AS RegType

InRegs.dx = 8 * (Row% - 1)
InRegs.cx = 8 * (Col% - 1)
InRegs.ax = 4
CALL INTERRUPT(&H33, Inregs, OutRegs)
```

Anche nel caso in cui venga richiesta una posizione che risulta essere esterna allo schermo, questo servizio non restituisce alcun messaggio di errore (ragione questa per cui *MouseMoveCursor( )* non produce nessun tipo di emissione) ma semplicemente posizionerà il puntatore sul bordo dello schermo.
Il listato 3.6 presenta l'intero sottoprogramma.

```
DECLARE SUB MouseMoveCursor (Row%, Col%)

TYPE RegType
   ax      AS INTEGER
   bx      AS INTEGER
   cx      AS INTEGER
   dx      AS INTEGER
   bp      AS INTEGER
   si      AS INTEGER
   di      AS INTEGER
   flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

SUB MouseMoveCursor (Row%, Col%)

   DIM InRegs AS RegType, OutRegs AS RegType

   InRegs.dx = 8 * (Row% - 1)
   InRegs.cx = 8 * (Col% - 1)
   InRegs.ax = 4
   CALL INTERRUPT(&H33, InRegs, OutRegs)

END SUB
```

Listato 3.6 *Il sottoprogramma MouseMoveCursor( ).*

SPOSTAMENTO DEL PUNTATORE ALL'ANGOLO SUPERIORE SINISTRO DELLO SCHERMO

Nell'esempio di seguito riportato, il puntatore del mouse viene posizionato nell'angolo superiore sinistro dello schermo, *(1, 1)*, dopo che è stato premuto un qualsiasi tasto.

```
REM Esempio di MouseMoveCursor

DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE SUB MouseHideCursor ( )
Check% = MouseInitialize%

LOCATE 1, 1
IF Check% = 0 THEN
   PRINT "Driver del mouse non installato."
```

```
ELSE
   PRINT "Mouse inizializzato."
   CALL MouseShowCursor
   LOCATE 10, 20
   PRINT "Premere un tasto per spostare il puntatore a (1, 1)."
   DO
   LOOP WHILE INKEY$ = " "
   CALL MouseMoveCursor (1, 1)
END IF
```

Come si potrà notare, l'uso del sottoprogramma *MouseMoveCursor( )* è abbastanza semplice poiché è sufficiente eseguire il trasferimento delle coordinate di riga e colonna desiderate. Naturalmente, la mancanza di messaggi di errore potrebbe essere considerata sia una cosa positiva che negativa. Da un lato i messaggi di errore spesso generano ansia, dall'altro, tuttavia, la loro presenza, se il puntatore viene erroneamente posizionato, potrebbe facilitare la comprensione dell'errore.

A questo punto, al sottoprogramma *MouseMoveCursor( )* potrebbe essere applicata una semplice modifica che controlli il numero di riga e colonna immesso. Nel caso in cui questi valori siano al di fuori dell'intervallo video ammesso si potrà sempre spostare il puntatore ma verrebbe anche emesso un messaggio di errore.

Il servizio successivo dell'interrupt &H33 che verrà preso in esame, il servizio 5, indica quante volte è stato premuto un determinato pulsante del mouse dall'ultima volta che è stato richiesto un rapporto di stato. L'uso di questo servizio, in sostanza, non obbliga a controllare la pressione di un tasto tutte le volte che questo viene premuto.

LETTURA DELLA "CODA" DEI PULSANTI PREMUTI

Potrebbe essere utile sviluppare un sottoprogramma che sia equivalente a *INKEY$*. In effetti è già stato presentato il sottoprogramma *MouseInformation( )*, ma questo prevedeva il controllo ogni volta che veniva premuto un pulsante. Indubbiamente, sarebbe molto più pratico poter memorizzare le varie operazioni del mouse e successivamente controllarle esclusivamente quando è specificatamente richiesto.

Il servizio 5 dell'interrupt &H33 consente di eseguire questa operazione poiché permette di leggere quante volte è stato premuto uno specifico pulsante dall'ultimo controllo effettuato. Inoltre, questo servizio restituisce anche la posizione di riga e colonna del puntatore relativa all'*ultima* pressione del tasto specificato. La pressione di un pulsante del mouse a volte è molto più significativa di quanto lo sia un semplice spostamento e, per questa ragione, questo servizio dovrebbe venire utilizzato come primaria routine di immissione di dati con il mouse.

Ora verrà realizzato un sottoprogramma a cui si assegnerà il nome *MouseTimesPressed( )* che collegherà il BASIC al servizio 5 dell'interrupt &H33. Il sottoprogramma dovrà essere richiamato quando si inizierà ad accettare l'immissione in modo da liberare la memoria buffer del mouse, attivando poi un loop che richiami ciclicamente il sottoprogramma per controllare quali e quante azioni sono state intraprese. La funzione di *MouseTimesPressed( )* è molto simile a *INKEY$*.

IL SOTTOPROGRAMMA MouseTimesPressed( )

Sarà ora necessario richiamare il servizio 5 per sapere quante volte è stato premuto uno specifico pulsante del mouse (sinistro o destro). Il valore restituito dovrebbe corrispondere a quante volte è stato premuto il pulsante e la posizione del puntatore relativa all'ultima volta in cui il pulsante è stato premuto. In altre parole, il sottoprogramma *MouseTimesPressed( )* dovrebbe essere impostato come segue:

```
CALL MouseTimesPressed (Button%, NumberTimes%, Row%, Col%)
```

La variabile *Button%* potrà essere impostata come l'interrupt &H33, per esempio, assegnandole un valore 0 quando viene premuto il pulsante sinistro e un valore 1 quando viene premuto il pulsante destro. In fase di restituzione si potrà ottenere una decodifica simile a quella di seguito riportata:

Variabile	*Descrizione*
NumberTimes%	Specifica quante volte è stato premuto un determinato pulsante del mouse dall'ultima volta che è stato invocato *MouseTimesPressed( )*.
Row%	Identifica la riga dello schermo (da 1 a 25) nella quale era posizionato il puntatore del mouse l'ultima volta che è stato premuto un qualsiasi pulsante.
Col%	Identifica la colonna (da 1 a 80) nella quale era posizionato il puntatore del mouse l'ultima volta che è stato premuto un qualsiasi pulsante.

L'uso dell'interrupt &H33, servizio 5, facilita notevolmente l'uso di questo sottoprogramma. Sarà sufficiente assegnare al registro *InRegs.bx* il valore relativo al pulsante premuto (0 per il pulsante destro e 1 per quello sinistro) e richiamare il servizio 5.

```
SUB MouseTimesPressed (Button%, NumberTimes%, Row%, Col%)

   DIM InRegs AS RegType, OutRegs AS RegType

   InRegs.bx = Button%
   InRegs.ax = 5
   CALL INTERRUPT(&H33, InRegs, OutRegs)
   :
   :
```

Il risultato verrà restituito dai registri *bx*, *cx* e *dx*. Il registro *bx* conterrà il numero di volte che è stato premuto uno dei pulsanti dl mouse (sinistro o destro) dall'ultima volta in cui è stato richiamato il servizio 5. Il registro *dx* conterrà il numero di riga (espresso in pixel) in cui era posizionato il puntatore al momento dell'ultima pressione di uno dei pulsanti del mouse e *cx* il numero della colonna (anch'esso espresso in pixel) corrispondente. Questi valori potranno essere quindi convertiti in numeri di riga e colonna (da 1 a 25 per le righe e da 1 a 80 per le colonne) per una loro più comprensibile interpretazione.

```
DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

SUB MouseTimesPressed (Button%, NumberTimes%, Row%, Col%)

   DIM InRegs AS RegType, OutRegs AS RegType

   InRegs.bx = Button%
   InRegs.ax = 5
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   NumberTimes% = OutRegs.bx
   Row% = OutRegs.dx \ 8 + 1
   Col% = OutRegs.cx \ 8 + 1
```

Il listato 3.7 presenta l'intero codice di *MouseTimesPressed( )*.

```
DECLARE SUB MouseTimesPressed (Button%, NumberTimes%, Row%, Col%)

REM Button% dovrà essere 0 per il pulsante sinistro e 1 per il
pulsante destro.

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)
```

continua

Listato 3.7 *Il sottoprogramma MouseTimesPressed( ).*

```
SUB MouseTimesPressed (Button%, NumberTimes%, Row%, Col%)
    DIM InRegs AS RegType, OutRegs AS RegType
    InRegs.bx = Button%
    InRegs.ax = 5
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    NumberTimes% = OutRegs.bx
    Row% = OutRegs.dx \ 8 + 1
    Col% = OutRegs.cx \ 8 + 1

END SUB
```

Listato 3.7 *Il sottoprogramma MouseTimesPressed( )*

CONTEGGIO DELLE PRESSIONI DI UN TASTO

Il programma che verrà ora presentato mostra l'esecuzione del sottoprogramma *MouseTimesPressed( )*. Il programma richiederà di premere varie volte il pulsante destro del mouse e in seguito di premere un qualsiasi tasto per generare un rapporto che riporti il conteggio di quante volte è stato premuto il pulsante, riportando, inoltre, la posizione del puntatore al momento dell'ultima pressione.

```
REM Esempio di MouseTimesPressed()

DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE SUB MouseTimesPressed (Button%, NumberTimes%, Row%, Col%)

Check% = MouseInitialize%

LOCATE 1, 1
IF Check% = 0 THEN
   PRINT "Driver del mouse non installato."
ELSE
   PRINT "Mouse inizializzato."
   CALL MouseShowCursor
   PRINT "Premere il pulsante destro varie volte"
   PRINT "e poi premere un tasto."
   DO
   LOOP WHILE INKEY$ = " "
   CALL MouseTimesPressed(1, NumberTimes%, Row%, Col%)
   PRINT "Il pulsante è stato premuto ";NumberTimes%;"volte."
   PRINT "All'ultima pressione il cursore era posizionato a"
   PRINT "(";Row%;","Col%;")."
END IF
```

Si osservi ora il programma sopra riportato. Innanzitutto il mouse è stato inizializzato tramite *MouseInitialize( )*, ne viene definita la posizione tramite *MouseShowCursor( )* e, infine vengono lette le informazioni del mouse tramite *MouseTimesPressed( )*.
L'unica limitazione è data dal fatto che *MouseTimesPressed( )* è in grado di restituire solo la posizione del puntatore relativa all'*ultima* pressione di un pulsante del mouse e sarà quindi necessario avviare periodicamente questo sottoprogramma per poterne controllare le posizioni intermedie (anche se non così spesso come accade per *MouseInformation( )*).
In pratica, questo significa che sarà necessario avviare relativamente spesso *MouseTimesPressed( )* in modo da avere sempre la possibilità di controllare e svuotare la coda relativa alle ultime operazioni a mouse avviate.
Un'altra limitazione potrebbe essere determinata dalla richiesta su quante volte è stato *rilasciato* un pulsante del mouse, e non quante volte esso è stato premuto. Per esempio, il rilascio di un pulsante del mouse è molto importante quando si richiederà di trascinare un oggetto da una parte all'altra dello schermo, oppure quando viene richiesto di attivare una selezione da un menu. Per fortuna, anche in questo caso, l'interrupt &H33 fornisce un servizio (il 6) adatto allo scopo.

LETTURA DELLA CODA DEI RILASCI DI UN PULSANTE DEL MOUSE

Il servizio 6 dell'interrupt &H33 potrà essere utilizzato per generare un sottoprogramma (*MouseTimesReleased( )*)che sia in grado di eseguire il conteggio delle volte in cui è stato *rilasciato* un pulsante del mouse. Questo sottoprogramma sarà in grado di fornire tutte le informazioni relative a quante volte è stato rilasciato un particolare pulsante del mouse a partire dal momento in cui è stato per l'ultima volta invocato il sottoprogramma oltre che indicare la posizione del puntatore all'interno dello schermo.

IL SOTTOPROGRAMMA MouseTimesReleased( )

Il programma *MouseTimesReleased( )* potrà essere invocato con:

```
CALL MouseTimesReleased (Button%, NumberTimes%, Row%, Col%)
```

Ancora una volta, la variabile *Button%* verrà impostata a 0 se si desiderano ottenere informazioni relative al pulsante sinistro e a 1 se le informazioni dovranno riportate lo stato del pulsante destro. Di seguito viene riportata una decodifica dei valori che verranno restituiti:

Variabile	*Tipo*	*Descrizione*
NumberTimes%	INTEGER	Numero dei rilasci di un determinato pulsante del mouse a partire dall'ultima invocazione di *MouseTimesReleased( )*.
Row%	INTEGER	Indica la riga (da 1 a 25) su cui era posizionato il puntatore all'*ultimo* rilascio del pulsante.
Col%	INTEGER	Indica la colonna (da 1 a 80) su cui era posizionato il puntatore all'*ultimo* rilascio del pulsante.

Questo sottoprogramma è molto simile a *MouseTimesPressed( )* con la sola eccezione che, in questo caso, verrà utilizzato il servizio 6 dell'interrupt &H33.

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.bx = Button%
InRegs.ax = 6
CALL INTERRUPT(&H33, InRegs, OutRegs)
:
:
```

Dopo l'invocazione l'informazione ottenuta dovrà essere decodificata esattamente come è stato fatto per *MouseTimesPressed( )*.

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.bx = Button%
InRegs.ax = 6
CALL INTERRUPT(&H33, InRegs, OutRegs)
NumberTimes% = OutRegs.bx
Row% = OutRegs.dx \ 8 + 1
Col% = OutRegs.cx \ 8 + 1
```

Il listato 3.8 riporta l'intero sottoprogramma.

```
DECLARE SUB MouseTimesReleased (Button%, NumberTimes%,
            Row%, Col%)

REM Button% dovrà essere 0 per il pulsante sinistro e 1 per
    il pulsante destro.

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
```

continua

Listato 3.8 *Il sottoprogramma MouseTimesReleased( )*

```
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

SUB MouseTimesReleased (Button%, NumberTimes%, Row%, Col%)

    DIM InRegs AS RegType, OutRegs AS RegType

    InRegs.bx = Button%
    InRegs.ax = 6
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    NumberTimes% = OutRegs.bx
    Row% = OutRegs.dx \ 8 + 1
    Col% = OutRegs.cx \ 8 + 1

END SUB
```

Listato 3.8 *Il sottoprogramma MouseTimesReleased( )*

CONTEGGIO DEI RILASCI DEL PULSANTE

Il prossimo programma è lo stesso utilizzato per *MouseTimesPressed( )* con la sola differenza che verranno conteggiti i rilasci di un pulsante e non le sue pressioni.

```
REM Esempio di MouseTimesReleased

DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE SUB MouseTimesReleased (Button%, NumberTimes%, Row%,
                                Col%)

Check% = MouseInitialize%

LOCATE 1, 1
IF Check% = 0 THEN
   PRINT "Driver del mouse non installato."
ELSE
   PRINT "Mouse inizializzato."
```

```
    CALL MouseShowCursor
    PRINT "Premere il pulsante destro varie volte"
    PRINT "e poi premere un tasto."
    DO
    LOOP WHILE INKEY$ = " "
CALL MouseTimesReleased(1, NumberTimes%, Row%, Col%)

PRINT "Il pulsante è stato rilasciato ";NumberTimes%;"volte."
PRINT "All'ultimo rilascio il cursore era posizionato a
(";Row%;","Col%;")."
END IF
```

Anche in questo caso si è prima dovuto inizializzare il mouse, in seguito controllare la posizione del puntatore e, dopo aver premuto un tasto, leggere le informazioni memorizzate in *MouseTimesReleased( )*.
Procedendo in questo capitolo verranno ora presentati altri servizi dell'interrupt &H33, prestando particolare attenzione a quelli che vengono ritenuti più interessanti. Per esempio, il prossimo servizio che verrà analizzato (il 7) consente di limitare l'operatività del puntatore a un determinato numero di righe e colonne.

> ***Consiglio***: Facendo uso anche del servizio 8, che restringe l'intervallo operativo delle righe, sarà possibile limitare l'operatività del puntatore a una specifica finestra attribuendo cosi ai programmi un aspetto davvero professionale.

LIMITAZIONE DEGLI SPOSTAMENTI ORIZZONTALI

Il servizio 7 dell'interrupt &H33 limita gli spostamenti orizzontali del cursore consentendo così di verificare azioni a mouse ristrette a un determinato intervallo di colonne. Verrà creato un sottoprogramma *MouseHorizontalRange( )* che potrà essere richiamato come segue:

```
CALL MouseHorizontalRange (Right%, Left%)
```

Questo programma restituirà i valori come di seguito riportato:

Nome parametro	*Descrizione*
Right%	Colonna destra dell'intervallo ammesso (da 1 a 80) entro cui si potrà spostare il puntatore.
Left%	Colonna sinistra dell'intervallo ammesso (da 1 a 80) entro cui si potrà spostare il puntatore.

IL SOTTOPROGRAMMA MouseHorizontalRange( )

Come si è già detto, l'interrupt &H33, servizio 7 consente di limitare gli spostamenti orizzontali del puntatore del mouse specificando, nel registro *cx*, una colonna limite di sinistra e, nel registro *dx*, una colonna limite di destra. Il valore, espresso in colonne, verrà convertito in pixel e verrà richiamato l'interrupt &H33.

```
DIM InRegs AS RegType, OutRegs AS RegType

InRegs.cx = 8 * (Right% - 1)
InRegs.dx = 8 * (Left% - 1)
InRegs.ax = 7
CALL INTERRUPT(&H33, InRegs, OutRegs)
:
:
```

Si tenga presente che il servizio 7 non restituisce alcun valore; il listato 3.9 presenta l'intero sottoprogramma.

```
DECLARE SUB MouseHorizontalRange (Right%, Left%)

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)
SUB MouseHorizontalRange (Right%, Left%)

    DIM InRegs AS RegType, OutRegs AS RegType

    InRegs.cx = 8 * (Right% - 1)
    InRegs.dx = 8 * (Left% - 1)
    InRegs.ax = 7
    CALL INTERRUPT(&H33, InRegs, OutRegs)

END SUB
```

Listato 3.9 *Il sottoprogramma MouseHorizontalRange( )*

UTILIZZO DI MouseHorizontalRange( )

Nell'esempio di seguito riportato lo spostamento del puntatore verrà limitato alla prima metà sinistra dello schermo impostando il limite della colonna sinistra a 1 e quello della colonna destra a 40.

```
REM Esempio di MouseHorizontalRange

DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE SUB MouseHorizontalRange (Right%, Left%)

Check% = MouseInitialize%

LOCATE 1, 1
IF Check% = 0 THEN
   PRINT "Driver del mouse non installato."
ELSE
   PRINT "Mouse inizializzato."
   CALL MouseShowCursor
   PRINT "Premere un tasto per limitare lo spostamento"
   PRINT "del puntatore alla prima metà sinistra dello schermo."
   DO
   LOOP WHILE INKEY$ = " "
CALL MouseHorizontalRange(1, 40)
END IF
```

***Nota**: Il servizio 7 ordina i due valori (e, come è facilmente comprensibile il valore della colonna sinistra dovrà essere inferiore a quello della colonna destra) perciò, non verrà generato alcun problema se MouseHorizontalRange( ) verrà richiamato nella forma MouseHorizontalRange(40, 1) oppure MouseHorizontalRange(1, 40).*

Se si preferisce utilizzare la modalità grafica, si potrà modificare *MouseHorizontal-Range( )* in modo che venga assunto un intervallo espresso in pixel.
Il successivo servizio dell'interrupt &H33 (il servizio 8) consentirà di limitare lo spostamento verticale del puntatore.

LIMITAZIONE DEGLI SPOSTAMENTI VERTICALI

Il complemento logico *MouseHorizontalRange( )* è, indubbiamente, *MouseVertical-Range( )*. Questo nuovo sottoprogramma limiterà gli spostamenti del puntatore a un determinato intervallo di righe (da 1 a 25) e potrà essere richiamato, per limitare lo spostamento del puntatore a una determinata finestra, utilizzando la seguente forma:

```
CALL MouseVerticalRange (Top%, Bottom%)
```

dove:

Nome parametro	*Descrizione*
Top%	corrisponde alla riga limite superiore dello spostamento (da 1 a 25);
Bottom%	corrisponde alla riga limite inferiore dello spostamento (da 1 a 25).

IL SOTTOPROGRAMMA MouseVerticalRange( )

In questo sottoprogramma verrà utilizzato il servizio 8 dell'interrupt &H33 in modo molto simile a quanto visto per il servizio 7, con l'eccezione che, in questo caso, la limitazione dello spostamento del puntatore del mouse verrà definita in base alle righe.

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.cx = 8 * (Top% - 1)
InRegs.dx = 8 * (Bottom% - 1)
InRegs.ax = 8
CALL INTERRUPT(&H33, InRegs, OutRegs)
:
:
```

Il listato 3.10 presenta l'intero sottoprogramma.

```
DECLARE SUB MouseVerticalRange (Top%, Bottom%)

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
```

continua

Listato 3.10 *Il sottoprogramma MouseVerticalRange( ).*

```
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

SUB MouseVerticalRange (Top%, Bottom%)

    DIM InRegs AS RegType, OutRegs AS RegType

    InRegs.cx = 8 * (Top% - 1)
    InRegs.dx = 8 * (Bottom% - 1)
    InRegs.ax = 8
    CALL INTERRUPT(&H33, InRegs, OutRegs)

END SUB
```

Listato 3.10 *Il sottoprogramma MouseVerticalRange( )*

UTILIZZO DI MouseVerticalRange( )

Il programma che segue restringerà gli spostamenti del puntatore del mouse alla prima metà superiore dello schermo richiamando *MouseVerticalRange(12, 1)*.

```
REM Esempio di MouseVerticalRange

DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE SUB MouseVerticalRange (Bottom%, Top%)

Check% = MouseInitialize%

LOCATE 1, 1
IF Check% = 0 THEN
   PRINT "Driver del mouse non installato."
ELSE
   PRINT "Mouse inizializzato."
   CALL MouseShowCursor
   PRINT "Premere un tasto per limitare lo spostamento"
   PRINT "del puntatore alla prima metà superiore dello schermo."
   DO
   LOOP WHILE INKEY$ = " "
CALL MouseVerticalRange(12, 1)
END IF
```

***Nota**: Come già detto per il servizio 7, anche il servizio 8 ordina i valori (il valore della riga limite superiore dovrà essere ovviamente più basso rispetto a quello della riga limita inferiore) perciò non importa l'ordine in cui vengono immessi i due valori: MouseVerticalRange(12, 1) è uguale a MouseVerticalRange(1, 12).*

L'interrupt &H33 contiene ancora un servizio (&HA) che consente di definire lo stile del puntatore del mouse.

IMPOSTAZIONE DEL PUNTATORE DEL MOUSE

Il servizio &HA (il 10) dell'interrupt &H33 consente di impostare lo stile del puntatore del mouse *esclusivamente* in modalità testo, assegnandogli uno qualsiasi dei caratteri ASCII disponibili. Per utilizzare questo servizio sarà necessario prima definire due "maschere": la *maschera video* e la *maschera puntatore*. Ogni maschera definisce sia il carattere che il byte di attributo dello stesso. Di seguito viene presentata una procedura che faciliterà la comprensione di questo concetto.

LA MASCHERA VIDEO E LA MASCHERA PUNTATORE

La *maschera video* determina quale parte del codice ASCII dovrà essere visualizzata quando il puntatore raggiunge una determinata posizione. Questa maschera viene sottoposta a un'operazione di *AND* logico con il codice ASCII del carattere sovrascritto e il relativo attributo video. Per esempio, se si desidera che il carattere rimanga inalterato sarà necessario applicare &HFF sia alla *maschera video del carattere* che alla *maschera video dell'attributo*. Per sovrascrivere completamente la posizione raggiunta dal cursore si dovrà assegnare alle maschere video del carattere e dell'attributo il valore *0*.

La *maschera del puntatore*, invece, determina come dovrà apparire il puntatore. La *maschera puntatore del carattere* e la *maschera puntatore dell'attributo* sono sottoposte a un'operazione di *XOR* logico con il risultato di un'operazione di *AND* logico tra il carattere in oggetto (e il relativo attributo) e la maschera video.

Il programma *MouseSetCursor( )* che verrà di seguito riportato è un esempio pratico di questa procedura. Il programma potrà essere richiamato nella seguente forma:

```
CALL MouseSetCursor (SMaskChar%, SMaskAttr%, CMaskChar%, CMaskAttr%)
```

Il significato dei parametri è il seguente:

Nome parametro	*Tipo*	*Descrizione*
SMaskChar%		Carattere ASCII della maschera video.
SMaskAttr%		Attributo della maschera video. Il carattere della maschera video e l'attributo sono di tipo AND rispetto al carattere e attributo già presenti a video. Per esempio, per sovrascrivere il carattere a video, impostare SMaskChar% = SMaskAttr% = 0. Per conservarlo, impostare SMask Char% = SMaskAttr% = &HFF.
CMaskChar%	INTEGER	Carattere ASCII per la maschera puntatore.
CMaskAttr%	INTEGER	Attributo della maschera puntatore. Il carattere della maschera puntatore e il relativo attributo sono di tipo XOR il cui risultato generato è di tipo AND in relazione al carattere su cui è posizionato il puntatore. Vale a dire che la maschera del puntatore determinerà l'aspetto del puntatore stesso.

Per esempio, per assegnare al puntatore un carattere ASCII particolare, sarà necessario sovrascrivere completamente il carattere esistente impostando *SMaskChar%* e *SMaskAttr%* a 0; è necessario poi caricare in *CMaskChar%* il codice ASCII del carattere che si desidera applicare al puntatore (come per esempio una freccia rivolta verso l'alto, corrispondente al codice ASCII 24) e definirne, infine, l'attributo (per esempio 7 per bianco su nero) da trasferire a *CmaskAttr%*.

***Consiglio**:* Se si desidera che il puntatore del mouse *inverta la polarità* del carattere su cui viene posizionato, sarà necessario impostare *SMaskChar%* a &HFF (che conserva il codice ASCII del carattere a video) e *SMaskAttr%* anch'esso a &HFF (che conserva intatto il byte di attributo). Impostare poi *CMaskAttr%* a &H77 per invertire l'attributo con XOR (utilizzare &H77 e non &HFF per evitare l'attivazione dei bit di lampeggiamento e alta intensità). Se si desidera invece limitarsi all'inversione di colore del carattere e non intervenire sul colore di sfondo impostare *CMaskAttr%* a 7.

Nel sottoprogramma che verrà creato verrà utilizzato il servizio &HA dell'interrupt &H33. Per impostare il puntatore del mouse sarà necessario caricare nel registro *bx*

il valore 0, nel registro *cx* l'intera maschera video (byte di attributo e relativo carattere), nel registro *dx* la maschera puntatore (byte attributo e relativo carattere). La difficoltà maggiore sarà determinata dal caricamento di questi due byte in *OutRges.cx* e in *OutRegs.dx*.

Tenendo presente come il BASIC gestisce i numeri negativi (si veda quanto descritto nel capitolo 11), non si potrà permettere che il primo bit di ordine superiore di un valore intero venga modificato in base al risultato di un'operazione matematica. Se verrà seguita questa procedura, il BASIC genererà un errore di *overflow*.

Questo, indubbiamente è un problema, poiché lo scopo è quello di caricare *SMaskAttr%* in *ch* (che corrisponde al byte alto di *InRegs.cx*) senza, tuttavia, poter definire *InRegs.cx = 2^8 * SMaskAttr%*. Se, per esempio, *SMaskAttr%* fosse uguale a &HFF, il primo bit di *InRegs.cx* verrebbe modificato e il BASIC genererebbe un errore di *overflow*. Per aggirare il problema si dovrà avviare una procedura, a volte noiosa, per impostare manualmente il primo bit di *InRegs.cx* e poi aggiungervi il resto degli altri bit.

Prima di tutto, sarà necessario controllare se il primo bit di *SMaskAttr%* (il bit che potrebbe generare problemi) è stato impostato. In caso affermativo, si dovrà impostare un valore intero (*TempS%*) nel quale sia stato impostato solo il primo bit.

```
DIM InRegs AS RegType, OutRegs AS RegType

REM Controlla eventuali overflow quando vengono caricati
    gli attributi nel byte di ordine superiore

IF SMaskAttr% AND 2^7 THEN
   TempS% = -1 XOR &H7FFF        'Imposta il primo bit
ELSE
   TempS% = 0
END IF
:
:
```

Il numero –1 viene memorizzato come &HFFFF in formato *INTEGER* (si rimanda, anche in questo caso, al capitolo 11). Per lasciare impostato solo il primo bit, questo dovrà essere sottoposto a un'operazione di XOR con &HFFFF. La stessa operazione dovrà essere eseguita con *InRegs.dx* e *CMaskAttr%* producendo così *TempC%* da assegnare alla maschera del puntatore, come di seguito riportato:

```
DIM InRegs AS RegType, OutRegs AS RegType

REM Controlla eventuali overflow quando vengono caricati
    gli attributi nel byte di ordine superiore

IF SMaskAttr% AND 2^7 THEN
   TempS% = -1 XOR &H7FFF        'Imposta il primo bit
ELSE
   TempS% = 0
```

```
END IF

IF CMaskAttr% AND 2^7 THEN
   TempC% = -1 XOR &H7FFF        'Imposta il primo bit
ELSE
   TempC% = 0
END IF
:
:
```

Infine, le maschere potranno essere caricate nei relativi registri come di seguito riportato:

```
DIM InRegs AS RegType, OutRegs AS RegType

REM Controlla eventuali overflow quando vengono caricati
    gli attributi nel byte di ordine superiore

IF SMaskAttr% AND 2^7 THEN
   TempS% = -1 XOR &H7FFF        'Imposta il primo bit
ELSE
   TempS% = 0
END IF

IF CMaskAttr% AND 2^7 THEN
   TempC% = -1 XOR &H7FFF        'Imposta il primo bit
ELSE
   TempC% = 0
END IF

InRegs.bx = 0
InRegs.cx = TempS% OR (256 * (SMaskAttr% AND &H7F) + SMaskChar%)
InRegs.dx = TempC% OR (256 * (CMaskAttr% AND &H7F) + CMaskChar%)
InRegs.ax = &HA
CALL INTERRUPT(&H33, InRegs, OutRegs)
:
:
```

Si noti che in questo modo si è aggirato il problema potenziale che poteva sorgere sul primo bit sia in *SMaskAttr%* che in *CMaskAttr%* sottoponendolo ad un'operazione di AND con &H7F prima di andare a operare sul byte di ordine superiore (cosa questa possibile moltiplicandolo per 2^8). È stato poi aggiunto il byte di ordine inferiore (*SMaskChar%* oppure *CMaskChar%*) e il risultato generato è stato utilizzato per generare i 15 bit inferiori di, rispettivamente, *InRegs.cx* o *InRegs.dx*. Dopo aver caricato questi registri, è stato invocato l'interrupt &H33, servizio &HA per impostare il puntatore del mouse. Il listato 3.11 presenta l'intero sottoprogramma.

```
DECLARE SUB MouseSetCursor (SMaskChar%, SMaskAttr%, CMaskChar%,⌉
                            CMaskAttr%)

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,⌉
                       OutRegs AS RegType)

SUB MouseSetCursor (SMaskChar%, SMaskAttr%, CMaskChar%,⌉
                    CMaskAttr%)

    DIM InRegs AS RegType, OutRegs AS RegType
    REM Controlla eventuali overflow quando vengono caricati
    REM gli attributi nel byte di ordine superiore

    IF SMaskAttr% AND 2^7 THEN
       TempS% = -1 XOR &H7FFF       'Imposta il primo bit
    ELSE
       TempS% = 0
    END IF

    IF CMaskAttr% AND 2^7 THEN
       TempC% = -1 XOR &H7FFF       'Imposta il primo bit
    ELSE
       TempC% = 0
    END IF

    InRegs.bx = 0
    InRegs.cx = TempS% OR (256 * (SMaskAttr% AND &H7F)⌉
                + SMaskChar%)
    InRegs.dx = TempC% OR (256 * (CMaskAttr% AND &H7F)⌉
                + CMaskChar%)
    InRegs.ax = &HA
    CALL INTERRUPT(&H33, InRegs, OutRegs)

END SUB
```

Listato 3.11 *Il sottoprogramma MouseSetCursor( ).*

TRASFORMAZIONE DEL PUNTATORE IN UN PUNTO

Si vedrà ora come poter mettere in opera il sottoprogramma *MouseSetCursor( )*.

```
REM Esempio di MouseSetCursor

   DECLARE FUNCTION MouseInitialize% ( )
   DECLARE SUB MouseSwowCursor ( )
   DECLARE SUB MouseSetCursor (ScMaskChar%, ScMaskAtr%,
                               CurMaskChar%, CurMaskAtr%)

   Check% = MouseInitialize%

LOCATE 1, 1
IF Check% = 0 THEN
   PRINT "Driver mouse non installato."
ELSE
   PRINT "Mouse inizializzato."
   CALL MouseShowCursor

   REM Trasforma il puntatore del mouse in un punto (ASCII 250)

   CALL MouseSetCursor(0, 0, 250, 7)
END IF
```

Il programma sopra riportato modifica semplicemente l'aspetto del puntatore assegnandogli un punto (per esempio, CHR$(250)) e posizionandolo al centro della posizione occupata dal carattere e con un attributo di tipo normale (per esempio, bianco su nero) definito dal valore 7.

> ***Consiglio***: Al puntatore del mouse potrà, se lo si desidera, essere applicato un diverso aspetto secondo quanto di seguito specificato.
>
> *Codice ASCII 24* = ↑
> *Codice ASCII 25* = ↓
> *Codice ASCII 26* = →
> *Codice ASCII 27* = ←

Si tenga tuttavia presente che sarà possibile utilizzare *MouseSetCursor( )* in modalità testo e la scelta dell'aspetto del puntatore sarà limitata a un unico carattere ASCII. La documentazione BASIC riporta un elenco completo dei caratteri ASCII disponibili. Ora verrà presa in considerazione la gestione del mouse con il BASIC PDS.

IL MOUSE CON IL BASIC PDS

La gestione del mouse con il BASIC Professional Development System (PDS) non è soddisfacente come ci si aspetterebbe da un programma simile.
Per iniziare, sarà necessario richiamare il sottoprogramma *MouseInit( )* per inizializzare il mouse (proprio come si è precedentemente fatto con *MouseInitialize( )*). Successivamente si potrà limitare lo spostamento del puntatore a un'area specifica dello schermo richiamando *MouseBorder(row1%, col1%, row2%, col2%)* che definisce i limiti dello spostamento descritti da *row1%, col1%* e *row2%, col2%* come illustrato in figura 3.1.

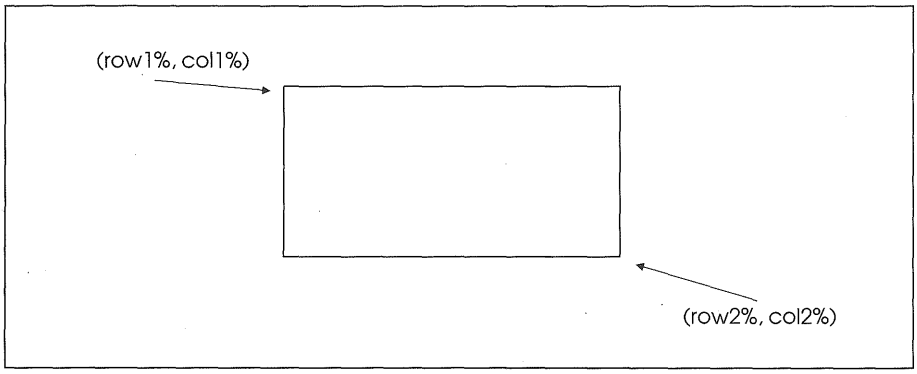

Figura 3.1

Dopo queste operazioni preliminari sarà possibile visualizzare il puntatore del mouse con *MouseShow( )* e nasconderlo nuovamente con *MouseHide( )*.
Per poter leggere le azioni del mouse si potrà ripetutamente richiamare il sottoprogramma *MousePoll( )* che opera in modo molto simile al programma *MouseInformation( )* precedentemente descritto, il che sta a significare che le azioni a mouse potranno essere controllate solo quando ve ne è una specifica necessità. Dopo *MousePoll( )* vi è ancora un'ultimo sottoprogramma da richiamare (*MouseDriver( )*) che consente di trasferire gli argomenti direttamente all'interrupt &H33 (accedendo così direttamente a tutti i servizi DOS ad esso collegati). Poiché questa operazione potrà essere facilmente eseguita con *INTERRUPT( )*, *MouseDriver( )* non sembra avere una grande utilità.
Verrà ora sviluppato un programma in BASIC PDS che restituisce la posizione del puntatore del mouse. Se si sta utilizzando QBX.EXE, al momento del suo caricamento, sarà necessario utilizzare l'opzione */L*. Si inizializzi innanzitutto il mouse:

```
DECLARE SUB MouseInit ( )

   CALL MouseInit
   :
```

Poi visualizzarne il puntatore:

```
DECLARE SUB MouseInit ( )
DECLARE SUB MouseShow ( )
   CALL MouseInit
   CALL MouseShow
   :
```

Ora si potrà utilizzare *MousePoll( )* che rimane in attesa della pressione di un pulsante del mouse. Le variabili che verranno completate sono: *row%, col%, lButton%* e *rButton%*. *lButton%* e *rButton%* restituiranno un valore *TRUE* o *FALSE* in relazione al fatto se il pulsante relativo è stato o meno premuto oltre che indicare la posizione del puntatore espressa in coordinate riga colonna (modalità testo).

```
DECLARE SUB MouseInit ( )
DECLARE SUB MousePoll (row%, col%, lButton%, rButton%)
DECLARE SUB MouseShow ( )
   CALL MouseInit
   CALL MouseShow
   PRINT "Il programma è in attesa della pressione"
   PRINT "del pulsante destro del mouse..."

DO
CALL MousePoll (row%, col%, lButton%, rButton%)
LOOP WHILE rButton% = FALSE
:
```

Dopo la pressione del pulsante del mouse si potrà ottenere come segue la posizione del puntatore:

```
DECLARE SUB MouseInit ( )
DECLARE SUB MousePoll (row%, col%, lButton%, rButton%)
DECLARE SUB MouseShow ( )

   CALL MouseInit
   CALL MouseShow

   PRINT "Il programma è in attesa della pressione"
   PRINT "del pulsante destro del mouse..."

   DO
   CALL MousePoll (row%, col%, lButton%, rButton%)
   LOOP WHILE rButton% = FALSE

PRINT "Il puntatore è posizionato a: "; row%; ","; col%; ")"
```

E con questo è terminata la descrizione della gestione del mouse con BASIC PDS. A differenza di quanto già espresso per le finestre, in questo caso le operazioni avviate sono relativamente semplici.

CONCLUSIONE

Questo capitolo ha preso in considerazione l'impostazione delle varie funzioni e sottoprogrammi per la gestione del mouse. Nel prossimo capitolo le procedure qui utilizzate verranno mese in pratica associandole al sistema di menu a tendina.

CAPITOLO 4

MENU A TENDINA

In questo capitolo verrà sviluppato un sistema di menu a tendina completo e funzionale, pronto per essere utilizzato in qualsiasi occasione. Si vedrà come creare una barra dei menu attivabile sia attraverso tastiera che via mouse. La creazione di questo sistema richiederà un piccolo sforzo, ma con quanto appreso nei precedenti capitoli riguardo finestre e mouse si otterrà un sistema dall'aspetto realmente professionale. Tutto quello che resta da fare è combinare i diversi moduli in modo da ottenere un programma funzionale.

***Nota**: Questo è un esempio di programmazione modulare.*

Il primo passo, come per qualsiasi tipo di creazione di programmi, è quello di avere in mente il risultato finale che si intende ottenere.

PROGETTAZIONE DI UN SISTEMA DI MENU A TENDINA

Prima di tutto sarà necessario considerare cosa si intende ottenere da un sistema di menu a tendina e come questo dovrà apparire a video. La *barra dei menu*, associata agli attributi di colore che ad essa verranno applicati, dovrebbe, in genere, apparire sulla parte superiore dello schermo e, se si prevede l'uso del mouse, si deve anche prendere in considerazione che per aprire un menu dovrà essere sufficiente cliccare (in genera con il pulsante sinistro del mouse) sulla relativa voce immessa sulla barra dei menu. Il menu generato verrà aperto al di sotto della voce selezionata, completo

di tutti gli attributi ad esso associati tenendo in considerazione la necessità di poter selezionare una qualunque delle voci in esso presenti. La figura 4.1 è un esempio classico di una barra dei menu dalla quale è stato aperto un menu a tendina.

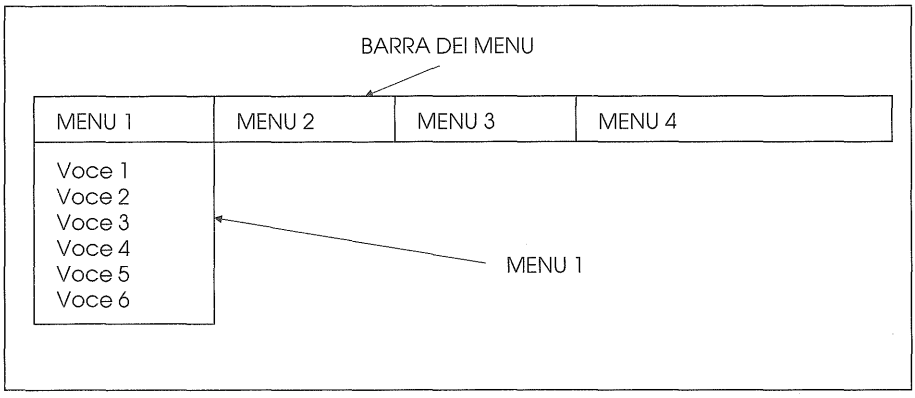

Figura 4.1

A questo punto l'utente può spostare il puntatore del mouse su una delle voci della barra dei menu, o su una delle voci del menu stesso, e cliccarvi sopra in modo che al programma venga indicato quale operazione avviare.
Inoltre, i menu dovrebbero essere operativi sia in modalità testo che in modalità grafica e le selezioni dovrebbero venire attivate sia con tastiera che con il mouse. Se non viene reperita la presenza del mouse, il programma dovrebbe essere in grado di passare automaticamente alla modalità operativa che prevede la sola tastiera per la selezione delle voci disponibili, prevedendo, in questo caso, la possibilità della pressione di una combinazione di tasti per avviare una particolare operazione. Una combinazione possibile potrebbe essere *ALT-**Tasto*** dove ***Tasto*** corrisponde alla prima lettera del menu da aprire nel quale verranno riportati i comandi per avviare una successiva operazione, anch'essi associati alla semplice pressione di uno o più tasti.
Si dovrebbe prevedere anche la possibilità di chiudere il menu se non si desidera avviare nessuno dei comandi in esso presenti e, in questo caso, la chiusura potrebbe essere generata dalla pressione, per esempio, di *ESC* (codifica questa adottata dalla maggior parte dei sistemi di menu a tendina), dall'ulteriore pressione di una diversa combinazione *ALT-**Tasto*** per l'apertura di un menu diverso da quello correntemente aperto oppure dalla pressione della combinazione *ALT-**Tasto*** precedentemente utilizzata per l'apertura del menu, che ne generi la sua chiusura.

CONSIDERAZIONI PARTICOLARI SUL SISTEMA A MENU

È necessario a questo punto considerare alcuni aspetti particolari relativi alla stesura del codice. Dal punto di vista della programmazione, il concetto principale è quello di creare un sistema a menu con il minimo sforzo possibile e, soprattutto, senza dover affrontare particolari problemi. Innanzitutto sarà necessario generare un sottoprogramma di inizializzazione del sistema a menu, a cui si assegnerà il nome *MenuInitialize( )*, che consenta l'impostazione delle voci da inserire all'interno della barra dei menu e all'interno dei menu stessi.

Per poter visualizzare la *barra dei menu* sarà necessario creare un successivo sottoprogramma (*MenuShow( )*) mentre, per nasconderla, si utilizzerà il sottoprogramma *MenuHide( )*.

Dopo queste operazioni, si passerà alla parte più complessa: come recuperare le informazioni relative alle varie selezioni effettuate all'interno della barra dei menu o all'interno dei menu stessi. La soluzione ottimale potrebbe essere quella di "far girare" il sistema dei menu indipendentemente dal resto del programma e interromperne l'esecuzione quando viene attivata una selezione. Purtroppo, questa possibilità non può essere gestita dal BASIC poiché esso non prevede la coesistenza di due diversi programmi in ambiente BASIC DOS.

La soluzione pratica, per cui, sarà quella di ottenere un sistema a menu che preveda *tutte* le immissioni possibili generate dal programma. In altre parole, ogni volta che si renderà necessaria un'immissione, sarà necessario richiamare una particolare funzione a cui assegnare, per esempio, il nome di *MenuGetEvent$( )*. L'operatività di questa funzione sarà molto simile a quella generata dalla contemporanea azione di *INKEY$*, *MouseTimesPressed( )*, *MouseTimesReleased( )* e dal programma di gestione dei menu. Nel caso in cui venga avviata un'operazione da tastiera, da mouse o strettamente collegata a un menu, la funzione *MenuGetEvent$( )* dovrebbe essere in grado di gestirla e memorizzarla (il che sta a significare un'operazione di controllo del contenuto della funzione relativamente frequente).

La funzione potrebbe essere impostata come di seguito riportato: se *MenuGetEvent$( )* restituisce una stringa composta da una o due caratteri (il o i caratteri corrispondenti ai tasti premuti) che verrà gestita esattamente come questa verrebbe gestita da *INKEY$*. Nel caso in cui, invece, venga avviata un'operazione a mouse, con pressione e successivo rilascio di uno dei suoi pulsanti, la funzione *MenuGetEvent$( )* dovrebbe memorizzare le coordinate video del puntatore del mouse in quel determinato momento. D'altra parte se la stringa corrisponde a una selezione di una voce di menu, questo sta a significare che è stata attivata una voce di un particolare menu. In questo caso la funzione *MenuGetEvent$( )* deve considerare da quale menu è stata avviata l'operazione indicando, nel contempo, la voce del menu che è stata selezionata.

È chiaro, quindi, che la funzione *MenuGetEvent$( )* deve assolutamente essere una delle routine primarie di immissione. L'intero processo si svilupperà in questa sequenza.

1. Invocazione di *MenuInitialize%( )* in modo da inizializzare il sistema a menu contenente i nomi dei vari menu e le relative voci all'interno dei menu a tendina che verranno generati.

2. Per attivare/disattivare la visualizzazione dei menu dovranno essere utilizzati, rispettivamente, i sottoprogrammi *MenuShow( )* e *MenuHide( )*.

3. La funzione *MenuGetEvent$( )* dovrà essere la routine di immissione primaria e questa dovrebbe essere una combinazione di *INKEY$* e della gestione del video a mouse con la presenza di un sistema di menu.

Come si è detto, la funzione *MenuGetEvent$( )* deve essere in grado di gestire tutti i tipi di immissione all'interno del programma provenienti sia da tastiera che da mouse.
Si passerà ora alla realizzazione pratica delle routine necessarie.

INIZIALIZZAZIONE DEL SISTEMA A MENU

Il primo passo per creare un sistema a menu è la generazione e l'uso della funzione *MenuInitialize%( )*. Si presupponga ora di dover generare un sistema a menu simile a quello riportato in figura 4.2.

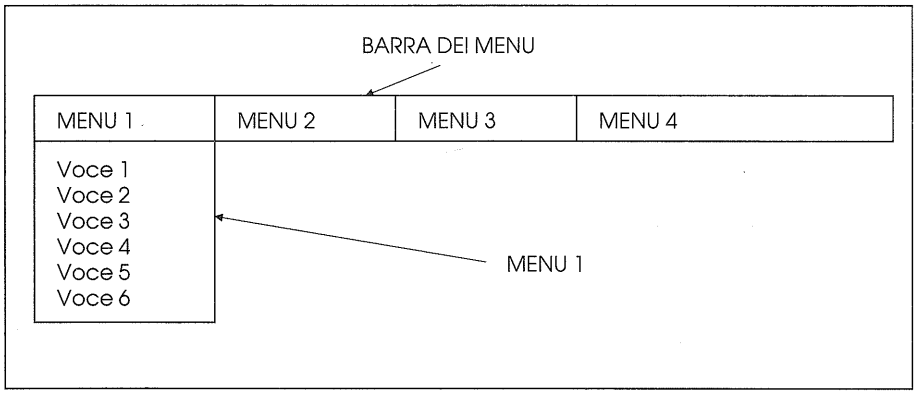

Figura 4.2

Come già detto, il primo passo è quello di inizializzare i nomi dei menu presenti sulla barra dei menu, le voci all'interno dei menu che verranno aperti e gli attributi video da applicare sia alla barra dei menu che ai menu a tendina che verranno via via generati.
Alla funzione *MenuInitialize%( )* dovranno essere innanzitutto trasferiti tutti i nomi che dovranno apparire sulla barra dei menu. Per poter trasferire questi nomi alla funzione sarà necessario utilizzare un array monodimensionale di tipo *STRING* a cui verrà assegnato il nome *MenuNames( )*. Per esempio, *MenuNames(1)* potrebbe essere definita come di seguito riportato:

```
MenuNames(1) = "Frutti"
```

per poter ottenere quanto riportato in figura 4.3.

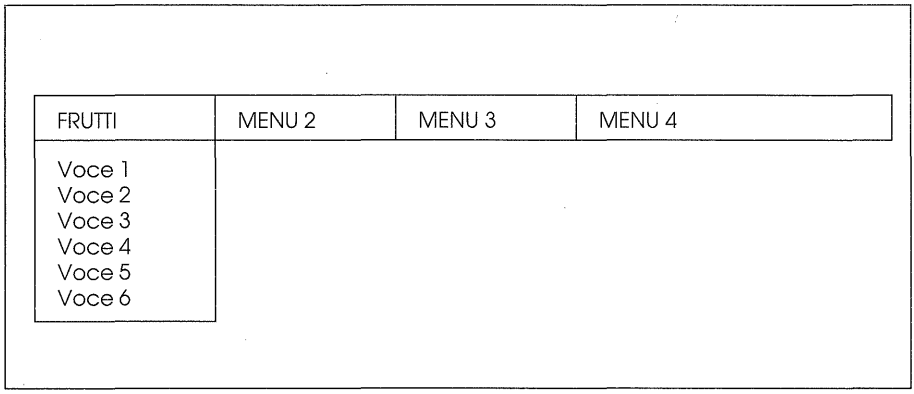

Figura 4.3

Il nome successivo sulla barra dei menu verrà associato a *MenuNames(2)*, il terzo a *MenuNames(3)* e così via.
Il limite imposto a *MenuNames( )* viene determinato dall'ampiezza dello schermo e dall'ampiezza di ciascun "campo" contenente un nome di menu. Un compromesso accettabile potrebbe essere la creazione di nomi di menu composti da otto caratteri e da un massimo di sette menu. *MenuInitialize%( )* potrà essere richiamata come di seguito riportato:

```
CALL MenuInitialize%(MenuNames( )...)
```

Sarà poi necessario trasferire al menu appena creato le relative voci interne. Queste potranno essere trasferite servendosi di un array bidimensionale (anch'esso di tipo *STRING*) a cui, per esempio, potrà essere assegnato il nome *MenuChoices( )*. Per esempio *MenuChoices(1, 1)* corrisponderà alla prima voce del menu 1, *MenuChoices(1, 2)* alla seconda voce del menu 1, e così via. Si presupponga ora di dover inserire nel menu "Frutti" cinque voci.

```
MenuChoices(1, 1) = "Mele"
MenuChoices(1, 2) = "Banane"
MenuChoices(1, 3) = "Uva"
MenuChoices(1, 3) = "Pesche"
MenuChoices(1, 4) = "Arance"
```

Il risultato del frammento sopra riportato dovrebbe generare quanto riportato in figura 4.4.

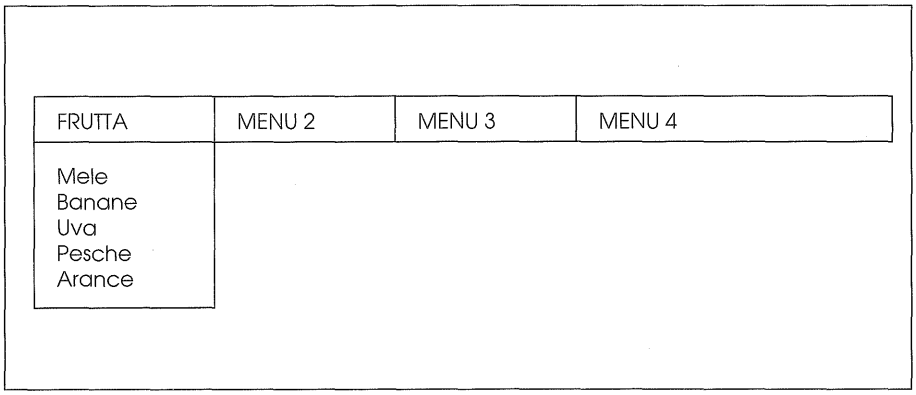

Figura 4.4

Si tenga presente che non tutti i menu hanno sempre lo stesso numero di voci: si presupponga di dover impostare il contenuto del menu 2, contenente solo tre voci.

```
MenuChoices(2, 1) = "Piselli"
MenuChoices(2, 2) = "Mais"
MenuChoices(2, 3) = "Broccoli"
```

Il risultato dovrebbe apparire simile a quanto riportato in figura 4.5.
Sarebbe opportuno, anche in questo caso porre un limite al contenuto del menu, per esempio, immettendo voci che non superino i 15 caratteri ciascuna e prevedendo un massimo di 24 voci all'interno di ciascun menu.
Di seguito viene riportato come immettere questo tipo di invocazione.

```
CALL MenuInitialize%(MenuNames( ), MenuChoices( )...)
```

Il successivo elemento che sarà necessario definire è ora l'attributo che si vorrà applicare alla barra dei menu e ai relativi menu che verranno generati da essa. Il trasferimento di un attributo alla barra dei menu è un'operazione relativamente semplice poiché sarà sufficiente trasferire a *BarAttrb%* un valore di tipo *INTEGER*. L'invocazione dovrebbe essere immessa come di seguito riportato.

```
CALL MenuInitialize%(MenuName( ), MenuChoices( ), BarAttrb%...)
```

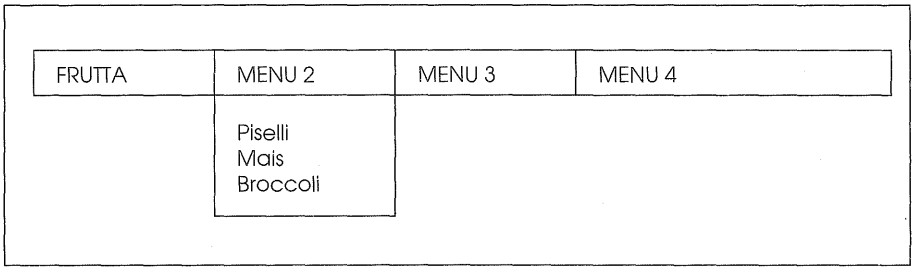

Figura 4.5

Infine, sarà necessario assegnare ai vari menu che verranno aperti gli attributi ad essi collegati. Questa operazione viene eseguita utilizzando un array monodimensionale a cui si potrebbe, per esempio, assegnare il nome di *MenuAttrbs( )*, contenente una serie di valori di tipo *INTEGER*. Per esempio, si può caricare *MenuAttrbs(3)* nel quale verranno definiti gli attributi da assegnare al menu 3 quando questo verrà aperto. Definire l'invocazione della funzione *MenuInitialize%( )* come di seguito riportato.

```
CALL MenuInitialize%(MenuNames( ), MenuChoices( ), BarAttrb%,
                     MenuAttrbs( ))
```

> ***Consiglio***: Potrebbe a volte accadere che l'attivazione di un'opzione di un determinato menu generi, a sua volta una nuova serie di menu da essa dipendenti, Per esempio, un menu potrebbe contenere una voce *File* che, se selezionata, genera a sua volta una nuova barra dei menu contenente, per esempio, *Legge, Registra, Apre, Chiude*. Per modificare la barra dei menu e le relative opzioni dei menu da essa generati, sarà necessario utilizzare nuovamente *MenuInitialize%( )*. Questa funzione potrà quindi essere richiamata per impostare la nuova serie di menu, insieme a *MenuShow( )*, *MenuHide( )* e a *MenuGetEvent$( )*.

LA FUNZIONE MenuInitialize%( )

Di seguito viene riportata la codifica degli elementi di *MenuInitialize%( )*.

Nome parametro	*Tipo*	*Descrizione*
MenuNames( )	STRING	Array di tipo monodimensionale contente le stringhe corrispondenti ai nomi dei menu che dovranno apparire sulla relativa bozza. Per esempio, *MenuNames(1)* potrebbe contenere *File* in modo che questa voce sia la prima ad apparire sulle

		barra dei menu. Si consiglia di limitare ogni stringa a un massimo di otto caratteri per poter creare una barra dei menu contenente sette diversi menu.
MenuChoices( )	STRING	Array bidimensionale contenente le stringhe relative alle varie voci inserite in un particolare menu. Per esempio, se *MenuChoices(1, 1)* corrisponde a *Apri file...* la prima voce del menu *File* sarà *Apri file...* Limitare la stringa a 15 caratteri e prevedere un massimo di 24 righe per ciascun menu (altrimenti il menu non potrebbe essere contenuto all'interno dello schermo).
BarAttrb%	INTEGER	Attributo video alla barra dei menu.
MenuAttrbs( )	INTEGER	Array monodimensionale contenente gli attributi video per ciascun menu che verrà generato. Per esempio, *MenuAttrbs(1)* imposterà l'attributo da applicare al primo menu.

MenuInitialize%( ) dovrebbe restituire *1* se l'impostazione dei menu risulta corretta e *0* in caso contrario. Come si è già visto per l'impostazione e la creazione delle finestre, anche in questo caso la procedura di creazione è relativamente lunga.

Innanzitutto sarà necessario impostare alcune istruzioni *COMMON*; si inizia con la dimensione di ciascun menu. Per rendere più semplice l'intera operazione, si potrà gestire ogni menu come se si trattasse di una finestra e adattarne alcuni dei codici di visualizzazione. Si consideri, in pratica, una serie di menu come se si trattasse di una serie di finestre. Si inizierà ora con la creazione del codice.

```
COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,⌉
              TopRow() AS INTEGER, TopCol() AS INTEGER,⌉
              BotRow() AS INTEGER, BotCol() AS INTEGER
              :
              :
```

Il passo successivo, sempre considerando ogni menu come una finestra, è quello di generare un array per ciascuno di essi. A *Text( )* verrà assegnato il contenuto della finestra (per esempio, le voci del menu, riga per riga), *OldText( )* conterrà il testo presente a video prima della visualizzazione del menu e *OldAttrb( )* gli attributi video di ogni posizione che verrà sovrascritta. Infine, sarà necessario generare un array (*Attribute( )*) contenente gli attributi del menu.

```
COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,
              TopRow() AS INTEGER, TopCol() AS INTEGER,
              BotRow() AS INTEGER, BotCol() AS INTEGER
COMMON SHARED /MenuB/ Attribute() AS INTEGER, Text() AS STRING,
              OldText() AS STRING, OldAttrb() AS STRING
   :
```

Quanto finora impostato per i menu potrà essere ugualmente applicato anche alla barra dei menu. Il testo della barra dei menu potrà essere memorizzato in una stringa a cui assegnare il nome *Bar$*; sarà anche necessario assegnare alla barra dei menu uno o più attributi, da trasferire a *BarAttrb*, conservando tuttavia in *OldBar$* il testo presente a video prima dell'apparizione della barra dei menu e in *OldBarAttrb$* gli attributi attivi prima della sovrascrittura.
È ora necessario creare altre due importanti variabili: la prima, a cui si potrà assegnare il nome *NumMenus%*, conterrà il numero dei nomi dei menu, la seconda (*MenusOnFlag%*) definisce l'attivazione/disattivazione del sistema dei menu.

```
COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,
              TopRow() AS INTEGER, TopCol() AS INTEGER,
              BotRow() AS INTEGER, BotCol() AS INTEGER
COMMON SHARED /MenuB/ Attribute() AS INTEGER, Text() AS STRING,
              OldText() AS STRING, OldAttrb() AS STRING
COMMON SHARED /MenuC/ Bar$, BarAttrb, MenusOnFlag%, NumMenus%,
              OldBar$, OldBarAttrb$
   :
   :
```

E con questo sono state impostate tutte le istruzioni *COMMON*. La vera e propria definizione di *MenuInitialize%( )* inizierà ora con un controllo preliminare dei limiti degli array liberando le necessarie allocazioni di memoria.

```
FUNCTION MenuInitialize% (MenuNames() AS STRING, MenuChoices()
         AS STRING, BarAttrb%, MenuAttrbs() AS INTEGER)
   MenuInitialize% = 0
   NumMenus% = UBOUND(MenuNames, 1)
   IF NumMenus% > 7 THEN EXIT FUNCTION
   IF UBOUND(MenuChoices, 2) > 24 THEN EXIT FUNCTION
   ERASE Rows
   ERASE Cols
   ERASE TopRow
   ERASE TopCol
   ERASE BotRow
   ERASE BotCol
   ERASE Attribute
   ERASE Text
   ERASE OldText
   ERASE OldAttrb
   Bar$ = ""
   MenusOnFlag% = 0
```

A questo punto, sarà necessario riempire le allocazioni di memoria precedentemente liberate. Si avvierà quindi un loop su ogni menu (definiti da *NumMenus%*) generando così la barra dei menu vera e propria.

```
FUNCTION MenuInitialize% (MenuNames() AS STRING, MenuChoices()
                AS STRING, BarAttrb%, MenuAttrbs() AS INTEGER)

   MenuInitialize% = 0

   NumMenus% = UBOUND(MenuNames, 1)
   IF NumMenus% > 7 THEN EXIT FUNCTION
   IF UBOUND(MenuChoices, 2) > 24 THEN EXIT FUNCTION

   ERASE Rows
   ERASE Cols
   ERASE TopRow
   ERASE TopCol
   ERASE BotRow
   ERASE BotCol
   ERASE Attribute
   ERASE Text
   ERASE OldText
   ERASE OldAttrb
   Bar$ = ""
   MenusOnFlag% = 0

   BarAttrb = BarAttrb%

   FOR i = 1 TO NumMenus%
      Bar$ = Bar$ + " " + MenuNames(i) + SPACE$(8 - LEN(MenuNames
               (i))) + CHR$(179)
      :
      :
   NEXT i
```

I nomi sono stati immessi sulla barra dei menu prelevandoli da *MenuNames( )* e vengono separati da un carattere corrispondente alla barra verticale (*CHR$(179)*). Il loop avviato su ciascun menu consente l'immissione le selezioni corrette prelevandole da *MenuChoices( )*.

Le voci immesse corrisponderanno alle selezioni avviabili da ciascun menu perciò sarà necessario memorizzarle nell'array *Text( )*, proprio come si è fatto quando si è lavorato sulle finestre. Su ogni menu sarà necessario avviare un loop che prenda in considerazione tutte le voci di *MenuChoices( )* definendo ciascuna di esse come una riga singola del menu (completandole eventualmente con spazi di riempimento).

```
FUNCTION MenuInitialize% (MenuNames() AS STRING, MenuChoices()
                AS STRING, BarAttrb%, MenuAttrbs() AS INTEGER)

   MenuInitialize% = 0

   NumMenus% = UBOUND(MenuNames, 1)
   IF NumMenus% > 7 THEN EXIT FUNCTION
   IF UBOUND(MenuChoices, 2) > 24 THEN EXIT FUNCTION

   ERASE Rows
   ERASE Cols
   ERASE TopRow
   ERASE TopCol
   ERASE BotRow
   ERASE BotCol
   ERASE Attribute
   ERASE Text
   ERASE OldText
   ERASE OldAttrb
   Bar$ = ""
   MenusOnFlag% = 0

   BarAttrb = BarAttrb%

   FOR i = 1 TO NumMenus%
      Bar$ = Bar$ + " " + MenuNames(i) + SPACE$(8 - LEN(MenuNames
               (i))) + CHR$(179)
      FOR j = 1 TO UBOUND(MenuChoices, 2)
         Temp$ = MenuChoices(i, j)
         IF Temp$ <> "" THEN
            IF LEN(Temp$) > 15 THEN EXIT FUNCTION
            Rows(i) = Rows(i) + 1
            Text(i, j) = " " + Temp$ + SPACE$(15 - LEN(Temp$))
         END IF
      NEXT j
      :
      :
   NEXT i
```

Infine, vi sono alcune altre variabili che dovranno essere impostate ogni volta che il loop viene avviato su ciascun menu. Sarà necessario registrare, fra l'altro, la riga iniziale e finale del menu (si ricordi che ciascun menu viene trattato come una finestra e, perciò, la riga iniziale di ciascun menu sarà la *2*— quella posta al di sotto della barra dei menu), le colonne limite destra e sinistra e gli attributi che verranno applicati al menu stesso. Dopo di ciò, necessario operare sulla barra dei menu in modo che questa sia composta da 80 caratteri (corrispondenti alla larghezza dello schermo).

```
FUNCTION MenuInitialize% (MenuNames() AS STRING, MenuChoices()
                AS STRING, BarAttrb%, MenuAttrbs() AS INTEGER)

   MenuInitialize% = 0

   NumMenus% = UBOUND(MenuNames, 1)
   IF NumMenus% > 7 THEN EXIT FUNCTION
   IF UBOUND(MenuChoices, 2) > 24 THEN EXIT FUNCTION

   ERASE Rows
   ERASE Cols
   ERASE TopRow
   ERASE TopCol
   ERASE BotRow
   ERASE BotCol
   ERASE Attribute
   ERASE Text
   ERASE OldText
   ERASE OldAttrb
   Bar$ = ""
   MenusOnFlag% = 0

   BarAttrb = BarAttrb%

   FOR i = 1 TO NumMenus%
      Bar$ = Bar$ + " " + MenuNames(i) + SPACE$(8 - LEN(MenuNames
               (i))) + CHR$(179)
      FOR j = 1 TO UBOUND(MenuChoices, 2)
         Temp$ = MenuChoices(i, j)
         IF Temp$ <> "" THEN
            IF LEN(Temp$) > 15 THEN EXIT FUNCTION
            Rows(i) = Rows(i) + 1
            Text(i, j) = " " + Temp$ + SPACE$(15 - LEN(Temp$))
         END IF
      NEXT j
      Cols(i) = 16
      TopRow(i) = 2
      TopCol(i) = 1 + 10 * (i - 1)
      BotRow(i) = 1 + Rows(i)
      BotCol(i) = TopCol(i) + 16
      Attribute(i) = MenuAttrbs(i)
   NEXT i

   Bar$ = Bar$ + SPACE$(80 - LEN(Bar$))

   MenuInitialize% = 1
```

Il listato 4.1 riporta l'intera composizione della funzione *MenuInitialize%( )*.

```
DECLARE FUNCTION MenuInitialize% (MenuNames() AS STRING,
                 MenuChoices() AS STRING, BarAttrb%, MenuAttrbs()
                 AS INTEGER)

    CONST MaxMenus = 7
    DIM Rows(1 TO MaxMenus) AS INTEGER
    DIM Cols(1 TO MaxMenus) AS INTEGER
    DIM TopRow(1 TO MaxMenus) AS INTEGER
    DIM TopCol(1 TO MaxMenus) AS INTEGER
    DIM BotRow(1 TO MaxMenus) AS INTEGER
    DIM BotCol(1 TO MaxMenus) AS INTEGER
    DIM Attribute(1 TO MaxMenus) AS INTEGER
    DIM Text(1 TO MaxMenus, 25) AS STRING
    DIM OldText(1 TO MaxMenus, 25) AS STRING
    DIM OldAttrb(1 TO MaxMenus, 25) AS STRING
    COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,
         TopRow() AS INTEGER, TopCol() AS INTEGER, BotRow()
         AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /MenuB/ Attribute() AS INTEGER, Text()
         AS STRING, OldText() AS STRING, OldAttrb() AS STRING
    COMMON SHARED /MenuC/ Bar$, BarAttrb, MenusOnFlag%, NumMenus%,
         OldBar$, OldBarAttrb$

FUNCTION MenuInitialize% (MenuNames() AS STRING, MenuChoices()
         AS STRING, BarAttrb%, MenuAttrbs() AS INTEGER)

    MenuInitialize% = 0

    NumMenus% = UBOUND(MenuNames, 1)
    IF NumMenus% > 7 THEN EXIT FUNCTION
    IF UBOUND(MenuChoices, 2) > 24 THEN EXIT FUNCTION

    ERASE Rows
    ERASE Cols
    ERASE TopRow
    ERASE TopCol
    ERASE BotRow
    ERASE BotCol
    ERASE Attribute
    ERASE Text
    ERASE OldText
    ERASE OldAttrb
    Bar$ = ""
    MenusOnFlag% = 0

    BarAttrb = BarAttrb%
```

continua

Listato 4.1 *La funzione MenuInitialize% per inizializzare il sistema a menu*

```
    FOR i = 1 TO NumMenus%
       Bar$ = Bar$ + " " + MenuNames(i) + SPACE$(8 - LEN⌉
                 (MenuNames(i))) + CHR$(179)
       FOR j = 1 TO UBOUND(MenuChoices, 2)
          Temp$ = MenuChoices(i, j)
          IF Temp$ <> "" THEN
             IF LEN(Temp$) > 15 THEN EXIT FUNCTION
             Rows(i) = Rows(i) + 1
             Text(i, j) = " " + Temp$ + SPACE$(15 - LEN(Temp$))
          END IF
       NEXT j
       Cols(i) = 16
       TopRow(i) = 2
       TopCol(i) = 1 + 10 * (i - 1)
       BotRow(i) = 1 + Rows(i)
       BotCol(i) = TopCol(i) + 16
       Attribute(i) = MenuAttrbs(i)
    NEXT i

    Bar$ = Bar$ + SPACE$(80 - LEN(Bar$))

    MenuInitialize% = 1

END FUNCTION
```

Listato 4.1 *La funzione MenuInitialize% per inizializzare il sistema a menu.*

È così completata l'intera procedura di creazione della funzione; è necessario, tuttavia, tenere presente che per visualizzare la barra dei menu sarà successivamente necessario invocare *MenuShow( )*.

PROVA DI INIZIALIZZAZIONE DI UNA BARRA DEI MENU

Di seguito viene riportata la procedura per la creazione di una barra dei menu contenente tre nomi.

```
DECLARE FUNCTION MenuInitialize% (MenuNames( ) AS STRING,⌉
        MenuChoices( ) AS STRING, BarAttrb%, MenuAttrbs( )⌉
        AS INTEGER)

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER
```

```
MenuNames(1) = "Frutta"
MenuChoices(1, 1) = "Mele"
MenuChoices(1, 2) = "Banane"
MenuChoices(1, 3) = "Uva"
MenuChoices(1, 4) = "Pesche"
MenuChoices(1, 5) = "Arance"

MenuNames(2) = "Verdura"
MenuChoices(2, 1) = "Piselli"
MenuChoices(2, 2) = "Grano"
MenuChoices(2, 3) = "Broccoli"

MenuNames(3) = "Carne"
MenuChoices(3, 1) = "Pollo"
MenuChoices(3, 2) = "Maiale"
MenuChoices(3, 3) = "Manzo"
MenuChoices(3, 4) = "Pesce"

FOR i= 1 TO 3
   MenuAttrbs(i) = &H61
NEXT i
BarAttrb% = &H24

Check% = MenuInitialize%(MenuNames( ), MenuChoices( ),
BarAttrb%, MenuAttrbs( ))
IF Check% = 0 THEN
   PRINT "ERRORE!"
ELSE
   PRINT "Menu impostato"
END IF
```

Come si può notare prima di invocare la funzione *MenuInitialize%( )* sono stati caricati tutti gli array; il passo successivo sarà quello di utilizzare *MenuShow( )* per visualizzare la barra dei menu.

VISUALIZZAZIONE DELLA BARRA DEI MENU

A questo punto si è pronti a creare la routine *MenuShow( )* che visualizzerà la barra dei menu nella parte superiore dello schermo. Inoltre questo sottoprogramma svuota la coda delle operazioni avviate con il mouse in modo che, quando viene avviata un'operazione sui menu, questa conterrà solo le azioni avviate dopo la visualizzazione della barra dei menu. Si noti che per rendere attiva la barra dei menu, dopo averla visualizzata, sarà necessario richiamare *MenuGetEvent$( )* che rimarrà in attesa di un'azione.

Prima di eseguire qualsiasi altra operazione, sarà necessario fare scomparire il puntatore del mouse dal video (riattivandone successivamente la visualizzazione). Questa operazione è necessaria per evitare eventuali operazioni di sovrascrittura del puntatore del mouse da parte della barra dei menu. Come detto nel precedente capitolo, se il puntatore del mouse verrà sovrascritto e in seguito lo si sposterà in altra zona dello schermo, il software di controllo del mouse ripristinerà l'aspetto *originale* del puntatore lasciando un "buco" all'interno della barra dei menu.

```
SUB MenuShow

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   REM Disattiva la visualizzazione del puntatore del mouse

   InRegs.ax = 2
   CALL INTERRUPT(&H33, InRegs, OutRegs)
   :
   :
```

***Nota**: Se non viene reperito il mouse o se il sistema mouse non è stato inizializzato, allora, quando il programma richiama nuovamente a video la visualizzazione del puntatore, l'operazione di riattivazione non avrà alcun effetto.*

Nel corpo del sottoprogramma *MenuShow( )* vi sono tre passi relativamente importanti. Il primo consiste nel salvataggio, tramite la funzione *SCREEN* del BASIC che consente di memorizzare sia i caratteri che gli attributi, del contenuto della regione dello schermo che verrà sovrascritta — vale a dire la prima riga dello schermo.

```
SUB MenuShow

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   REM Disattiva la visualizzazione del puntatore del mouse

   InRegs.ax = 2
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   REM Salva la precedente prima riga dello schermo

      temp1$ = ""
      temp2$ = ""
```

```
FOR j = 1 TO 80
   temp1$ = temp1$ + CHR$(SCREEN(1, j))
   temp2$ = temp2$ + CHR$(SCREEN(1, j, 1))
NEXT j
OldBar$ = temp1$
OldBarAttrb$ = temp2$
:
:
```

La seconda consente di visualizzare successivamente la barra dei menu, utilizzando l'interrupt &H10, servizio 9, esattamente come questo è stato utilizzato da *WindowShow( )*.

```
SUB MenuShow

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   REM Disattiva la visualizzazione del puntatore del mouse

   InRegs.ax = 2
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   REM Salva la precedente prima riga dello schermo

      temp1$ = ""
      temp2$ = ""
      FOR j = 1 TO 80
         temp1$ = temp1$ + CHR$(SCREEN(1, j))
         temp2$ = temp2$ + CHR$(SCREEN(1, j, 1))
      NEXT j
      OldBar$ = temp1$
      OldBarAttrb$ = temp2$

   REM Visualizza la barra dei menu

   InRegs.cx = 1

      FOR j = 1 TO 80
         LOCATE 1, j
         InRegs.ax = &H900 + ASC(MID$(Bar$, j, 1))
         InRegs.bx = BarAttrb
         CALL INTERRUPT(&H10, InRegs, OutRegs)
      NEXT j
      :
      :
```

Infine, è necessario preparare l'ambiente per il resto del sistema a menu. Riattivare quindi la visualizzazione del puntatore del mouse, indicando così che è stata attivata la barra dei menu — insieme all'intero sistema a menu (impostando *MenuOnFlag%* a 1) —, riportare il puntatore del mouse alla sua posizione originale e svuotare la coda delle precedenti operazioni eseguita con il mouse.

```
SUB MenuShow

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   REM Disattiva la visualizzazione del puntatore del mouse

   InRegs.ax = 2
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   REM Salva la precedente prima riga dello schermo

      temp1$ = ""
      temp2$ = ""
      FOR j = 1 TO 80
         temp1$ = temp1$ + CHR$(SCREEN(1, j))
         temp2$ = temp2$ + CHR$(SCREEN(1, j, 1))
      NEXT j
      OldBar$ = temp1$
      OldBarAttrb$ = temp2$

   REM Visualizza la barra dei menu

   InRegs.cx = 1

      FOR j = 1 TO 80
         LOCATE 1, j
         InRegs.ax = &H900 + ASC(MID$(Bar$, j, 1))
         InRegs.bx = BarAttrb
         CALL INTERRUPT(&H10, InRegs, OutRegs)
      NEXT j

   REM Riattiva la visualizzazione del puntatore del mouse

   InRegs.ax = 1
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   REM Visualizza la barra dei menu.

   MenusOnFlag% = 1
```

```
LOCATE CurRow%, CurCol%

REM Svuota la coda delle operazioni a mouse

InRegs.bx = 0
InRegs.ax = 5
CALL INTERRUPT(&H33, InRegs, OutRegs)
InRegs.bx = 1
InRegs.ax = 5
CALL INTERRUPT(&H33, InRegs, OutRegs)
InRegs.bx = 0
InRegs.ax = 6
CALL INTERRUPT(&H33, InRegs, OutRegs)
InRegs.bx = 1
InRegs.ax = 6
CALL INTERRUPT(&H33, InRegs, OutRegs)
```

A questo punto, la prima riga dello schermo dovrebbe riportare la barra dei menu precedentemente creata. Il listato 4.2 presenta l'intero sottoprogramma.

```
DECLARE SUB MenuShow ()

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

    CONST MaxMenus = 7
    DIM Rows(1 TO MaxMenus) AS INTEGER
    DIM Cols(1 TO MaxMenus) AS INTEGER
    DIM TopRow(1 TO MaxMenus) AS INTEGER
    DIM TopCol(1 TO MaxMenus) AS INTEGER
    DIM BotRow(1 TO MaxMenus) AS INTEGER
    DIM BotCol(1 TO MaxMenus) AS INTEGER
    DIM Attribute(1 TO MaxMenus) AS INTEGER
```

continua

Listato 4.2 *Il sottoprogramma MenuShow( ) per la visualizzazione della barra dei menu*

```
    DIM Text(1 TO MaxMenus, 25) AS STRING
    DIM OldText(1 TO MaxMenus, 25) AS STRING
    DIM OldAttrb(1 TO MaxMenus, 25) AS STRING
    COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,
         TopRow() AS INTEGER, TopCol() AS INTEGER, BotRow()
         AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /MenuB/ Attribute() AS INTEGER, Text()
         AS STRING, OldText() AS STRING, OldAttrb() AS STRING
    COMMON SHARED /MenuC/ Bar$, BarAttrb, MenusOnFlag%,
         NumMenus%, OldBar$, OldBarAttrb$

SUB MenuShow

    DIM InRegs AS RegType, OutRegs AS RegType

    CurRow% = CSRLIN
    CurCol% = POS(0)

    REM Disattiva la visualizzazione del puntatore del mouse

    InRegs.ax = 2
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    REM Salva la precedente prima riga dello schermo

       temp1$ = ""
       temp2$ = ""
       FOR j = 1 TO 80
          temp1$ = temp1$ + CHR$(SCREEN(1, j))
          temp2$ = temp2$ + CHR$(SCREEN(1, j, 1))
       NEXT j
       OldBar$ = temp1$
       OldBarAttrb$ = temp2$

    REM Visualizza la barra dei menu
    InRegs.cx = 1

       FOR j = 1 TO 80
          LOCATE 1, j
          InRegs.ax = &H900 + ASC(MID$(Bar$, j, 1))
          InRegs.bx = BarAttrb
          CALL INTERRUPT(&H10, InRegs, OutRegs)
       NEXT j

    REM Riattiva la visualizzazione del puntatore del mouse
```

continua

Listato 4.2 *Il sottoprogramma MenuShow( ) per la visualizzazione della barra dei menu*

```
    InRegs.ax = 1
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    REM Visualizza la barra dei menu.

    MenusOnFlag% = 1

    LOCATE CurRow%, CurCol%

    REM Svuota la coda delle operazioni a mouse

    InRegs.bx = 0
    InRegs.ax = 5
    CALL INTERRUPT(&H33, InRegs, OutRegs)
    InRegs.bx = 1
    InRegs.ax = 5
    CALL INTERRUPT(&H33, InRegs, OutRegs)
    InRegs.bx = 0
    InRegs.ax = 6
    CALL INTERRUPT(&H33, InRegs, OutRegs)
    InRegs.bx = 1
    InRegs.ax = 6
    CALL INTERRUPT(&H33, InRegs, OutRegs)

END SUB
```

Listato 4.2 *Il sottoprogramma MenuShow( ) per la visualizzazione della barra dei menu*

PROVA DI VISUALIZZAZIONE DELLA BARRA DEI MENU

Si potrà nuovamente utilizzare l'esempio di *MenuInitialize%( )* alla fine del quale verrà riportato *MenuShow( )* che consente la visualizzazione della barra dei menu.

```
DECLARE FUNCTION MenuInitialize% (MenuNames( ) AS STRING,
        MenuChoices( ) AS STRING, BarAttrb%, MenuAttrbs( )
        AS INTEGER)
DECLARE SUB MenuShow( )

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
```

```
MenuChoices(1, 2) = "Banane"
MenuChoices(1, 3) = "Uva"
MenuChoices(1, 4) = "Pesche"
MenuChoices(1, 5) = "Arance"

MenuNames(2) = "Verdura"
MenuChoices(2, 1) = "Piselli"
MenuChoices(2, 2) = "Grano"
MenuChoices(2, 3) = "Broccoli"

MenuNames(3) = "Carne"
MenuChoices(3, 1) = "Pollo"
MenuChoices(3, 2) = "Maiale"
MenuChoices(3, 3) = "Manzo"
MenuChoices(3, 4) = "Pesce"

FOR i= 1 TO 3
   MenuAttrbs(i) = &H61
NEXT i
BarAttrb% = &H24

Check% = MenuInitialize%(MenuNames( ), MenuChoices( ),
          BarAttrb%, MenuAttrbs( ))
IF Check% = 0 THEN
   PRINT "ERRORE!"
ELSE
   PRINT "Menu impostato"
END IF

CALL MenuShow
```

A video dovrebbe ora essere visibile (anche se non ancora attiva) la barra dei menu. Notare che, quando si avviano operazioni di scorrimento del contenuto del video, la barra dei menu potrebbe scomparire dallo schermo. È questo un aspetto particolarmente importante da tenere presente anche se il richiamo a video della barra dei menu potrà essere avviato semplicemente richiamando *MenuShow( )*. Questa operazione, tuttavia, genera una barra dei menu con un aspetto diverso rispetto a quella originale perciò, in genere, i programmi utilizzano una tecnica di scorrimento di tipo avanzato oppure evitano qualsiasi tipo di possibilità di scorrimento.

Consiglio: Anche se complicato, il linguaggio assembly è in grado di controllare i tentativi avviati da BASIC per fare scorrere il contenuto dello schermo. Sarà sempre possibile scrivere un programma *residente* in grado di intercettare i richiamo all'interrupt &H10, servizi 6 e 7, del BIOS, i quali controllano lo scorrimento dello schermo. Per ulteriori dettagli si veda l'appendice.

A questo punto la barra dei menu dovrebbe essere presente a video e il passo successivo sarà quello di farla scomparire.

RIMOZIONE DELLA BARRA DEI MENU

Il sottoprogramma che viene ora presentato, *MenuHide( )*, è il logico complemento di *MenuShow( )* ed è quello che consente, quando necessario, di fare scomparire da video la barra dei menu.

> ***Consiglio***: Il sottoprogramma *MenuHide( )* trova una sua logica collocazione alla fine di un programma, quando sarà necessario ripristinare lo schermo.

Lo scopo finale di questo sottoprogramma è quello di leggere il contenuto originale della prima riga dello schermo dalle istruzioni *COMMON* e di rimpiazzarlo (dopo aver nascosto il puntatore del mouse) senza, tuttavia, doverlo memorizzare poiché questo è già in memoria (nella stringa *Bar$*).
La prima operazione da compiere è quella di salvare la posizione del puntatore e disattivarne la visualizzazione.

```
SUB MenuHide

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   REM Disattiva il cursore del mouse

   InRegs.ax = 2
   CALL INTERRUPT(&H33, InRegs, OutRegs)
   :
   :
```

Ora sarà necessario ripristinare il testo precedentemente visualizzato.

```
SUB MenuHide

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   REM Disattiva il cursore del mouse
```

```
InRegs.ax = 2
CALL INTERRUPT(&H33, InRegs, OutRegs)

REM Visualizza il testo precedente

InRegs.cx = 1

   FOR j = 1 TO 80
      LOCATE 1, j
      InRegs.ax = &H900 + ASC(MID$(OldBar$, j, 1))
      InRegs.bx = ASC(MID$(OldBarAttrb$, j, 1))
      CALL INTERRUPT(&H10, InRegs, OutRegs)
   NEXT j
:
:
```

Per terminare si dovrà indicare che la barra dei menu deve essere disattivata (questa operazione, naturalmente disattiva anche l'intero sistema a menu e le immissioni verranno trasferite al programma chiamante) impostando *MenusOnFlag%* a 0. Verrà successivamente ripristinata la visualizzazione del puntatore del mouse riportandolo alla posizione che esso occupava quando è stato richiamato il sottoprogramma *MenuHide( )*.

```
SUB MenuHide

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   REM Disattiva il cursore del mouse

   InRegs.ax = 2
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   REM Visualizza il testo precedente

   InRegs.cx = 1

      FOR j = 1 TO 80
         LOCATE 1, j
         InRegs.ax = &H900 + ASC(MID$(OldBar$, j, 1))
         InRegs.bx = ASC(MID$(OldBarAttrb$, j, 1))
         CALL INTERRUPT(&H10, InRegs, OutRegs)
      NEXT j

   REM Riattiva la visualizzazione del mouse
```

```
InRegs.ax = 1
CALL INTERRUPT(&H33, InRegs, OutRegs)

REM Disattiva la visualizzazione della barra dei menu

MenusOnFlag% = 0

LOCATE CurRow%, CurCol%
:
:
```

Il listato 4.3 mostra l'intero sottoprogramma.

```
DECLARE SUB MenuHide ()
TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE
DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,⌉
                       OutRegs AS RegType)

    CONST MaxMenus = 7
    DIM Rows(1 TO MaxMenus) AS INTEGER
    DIM Cols(1 TO MaxMenus) AS INTEGER
    DIM TopRow(1 TO MaxMenus) AS INTEGER
    DIM TopCol(1 TO MaxMenus) AS INTEGER
    DIM BotRow(1 TO MaxMenus) AS INTEGER
    DIM BotCol(1 TO MaxMenus) AS INTEGER
    DIM Attribute(1 TO MaxMenus) AS INTEGER
    DIM Text(1 TO MaxMenus, 25) AS STRING
    DIM OldText(1 TO MaxMenus, 25) AS STRING
    DIM OldAttrb(1 TO MaxMenus, 25) AS STRING
    COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,⌉
         TopRow() AS INTEGER, TopCol() AS INTEGER, BotRow()⌉
         AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /MenuB/ Attribute() AS INTEGER, Text()⌉
         AS STRING, OldText() AS STRING, OldAttrb() AS STRING
    COMMON SHARED /MenuC/ Bar$, BarAttrb, MenusOnFlag%,⌉
         NumMenus%, OldBar$, OldBarAttrb$
```

continua

Listato 4.3 *Il sottoprogramma MenuHide( ) per nascondere la barra dei menu*

```
SUB MenuHide

    DIM InRegs AS RegType, OutRegs AS RegType

    CurRow% = CSRLIN
    CurCol% = POS(0)

    REM Disattiva il cursore del mouse
    InRegs.ax = 2
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    REM Visualizza il testo precedente

    InRegs.cx = 1

       FOR j = 1 TO 80
          LOCATE 1, j
          InRegs.ax = &H900 + ASC(MID$(OldBar$, j, 1))
          InRegs.bx = ASC(MID$(OldBarAttrb$, j, 1))
          CALL INTERRUPT(&H10, InRegs, OutRegs)
       NEXT j

    REM Riattiva la visualizzazione del mouse

    InRegs.ax = 1
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    REM Disattiva la visualizzazione della barra dei menu

    MenusOnFlag% = 0

    LOCATE CurRow%, CurCol%

END SUB
```

Listato 4.3 *Il sottoprogramma MenuHide( ) per nascondere la barra dei menu.*

PROVA DI RIMOZIONE DELLA BARRA DEI MENU

L'esempio di seguito riportato visualizza la barra dei menu e in seguito, alla pressione di un tasto qualsiasi, la rimuove.

```
DECLARE FUNCTION MenuInitialize% (MenuNames( ) AS STRING,
        MenuChoices( ) AS STRING, BarAttrb%, MenuAttrbs( )
        AS INTEGER)
```

```
DECLARE SUB MenuShow( )
DECLARE SUB MenuHide( )

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
   MenuChoices(2, 1) = "Piselli"
   MenuChoices(2, 2) = "Grano"
   MenuChoices(2, 3) = "Broccoli"

   MenuNames(3) = "Carne"
   MenuChoices(3, 1) = "Pollo"
   MenuChoices(3, 2) = "Maiale"
   MenuChoices(3, 3) = "Manzo"
   MenuChoices(3, 4) = "Pesce"

   FOR i= 1 TO 3
      MenuAttrbs(i) = &H61
   NEXT i
   BarAttrb% = &H24

   Check% = MenuInitialize%(MenuNames( ), MenuChoices( ),
        BarAttrb%, MenuAttrbs( ))
   IF Check% = 0 THEN
      PRINT "ERRORE!"
   ELSE
      PRINT "Menu impostato"
   END IF

   CALL MenuShow

   PRINT "Premere un tasto per rimuovere la barra dei menu."
   DO
   LOOP WHILE INKEY$ = " "

   CALL MenuHide( )
```

L'esempio sopra riportato inizializza la barra dei menu e ogni menu da essa dipendente visualizzandone il contenuto e in seguito rimuovendolo. Il prossimo passo sarà quello di trasferire le immissioni con *MenuGetEvent$( )*.
Il richiamo di *MenuGetEvent$( )* consente di attivare la barra dei menu. Quando questa funzione viene richiamata, il sistema rimarrà in attesa di un'immissione; si ricorda che, poiché la funzione è in grado di ricevere diversi tipi di immissione, essa potrebbe diventare davvero "gigantesca".

LETTURA DELLE IMMISSIONI DA MENU

Ora si dovrà impostare un programma di immissione (da tastiera, da mouse e da menu). Dopo aver inizializzato il sistema a menu e aver visualizzato la barra dei menu sarà necessario invocare la funzione *MenuGetEvent$( )* per poter immettere i dati necessari. Questa funzione rimarrà in attesa della pressione di un tasto, della pressione di un pulsante del mouse oppure della selezione di una voce di menu restituendo, infine, il risultato dell'operazione avviata.

***Nota**: Se si desidera che la funzione non rimanga in attesa di un'immissione sarà necessario scrivere un'ulteriore funzione, MenuCheckEvent$( ), la cui funzionalità, all'interno del sistema a menu, sarà la stessa di quella avviata da INKEY$. Se il programma rimane in attesa di un'azione da tastiera, da mouse o di una selezione da menu, questa condizione verrà memorizzata. In caso contrario, verrà immediatamente restituito un valore pari a " ", proprio come verrebbe richiesto da INKEY$.*

Poiché lo scopo di *MenuGetEvent$( )* è quello di registrare le operazioni a tastiera, a mouse o a menu, la sua definizione dovrebbe essere come quella di seguito riportata:

```
InString$ = MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,
                           ScCol%)
```

Notare che le variabili che verranno trasferite a *MenuGetEvent$( )* dovranno contenere, a seconda della stringa che verrà restituita, quanto di seguito riportato.

InString$	*Descrizione*
STRING di LEN 1	Pressione di un singolo carattere. La lettura di questa immissione è identica a quella che viene eseguita da *INKEY$*.
STRING di LEN 2	Pressione di un carattere con codice ASCII di tipo esteso. La lettura di questa immissione è identica a quella che viene eseguita da *INKEY$*.

InString$	*Descrizione*
"mousedown"	Indica quale pulsante del mouse è stato premuto: *Button% = 0* indica il pulsante sinistro, *Button% = 1* indica il pulsante destro. Le coordinate video (*ScRow%* e *ScCol%*) vengono espresse con un valore di righe e colonne video (*1-25* e *1-80*).
"mouseup"	Indica quale pulsante del mouse è stato rilasciato: *Button% = 0* indica il pulsante sinistro, *Button% = 1* indica il pulsante destro. Le coordinate video (*ScRow%* e *ScCol%*) vengono espresse con un valore di righe e colonne video (*1-25* e *1-80*).
"menuopen"	Indica il menu che è stato aperto: numero del menu definito da *MenuNo%*.
"menuclose"	Indica il menu che è stato chiuso: numero del menu definito da *MenuNo%*.
"menuchoice"	Indica la selezione di menu. *MenuNo% = numero del menu*; *ChoiceNo% = numero dell'opzione del menu*. I numeri dei menu e delle relative opzioni corrispondono agli indici degli array che sono stati trasferiti a *MenuInitialize%( )*.

Se non si ritiene necessario registrare le azioni generate da *mouseup, mousedown, menuopen* o *menuclose* sarà sufficiente richiamare nuovamente *MenuGetEvent$( )* fino a quando non verrà letta la pressione di un tasto oppure fino a quando non verrà restituita una stringa corrispondente a un'opzione di menu.
In questo caso, il numero del menu selezionato verrà prelevato da *MenuNo%* e il numero della selezione dall'interno del menu verrà prelevata da *ChoiceNo%*.

```
"menuchoice" = MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%,
                ScRow%, ScCol%)
```

Questi numeri corrisponderanno agli indici degli array che sono stati trasferiti a *MenuInitialize%( )*. Per esempio, se il menu 1 è stato inizializzato come di seguito riportato:

```
MenuChoices(1, 1) = "Mele"
MenuChoices(1, 2) = "Banane"
MenuChoices(1, 3) = "Uva"
MenuChoices(1, 4) = "Pesche"
MenuChoices(1, 5) = "Arance"
```

e *MenuGetEvent$( )* restituisce un valore di *MenuNo% = 1* e *ChoiceNo% = 4*, allora questo significherà che la selezione avviata corrisponderà a *MenuChoices(1, 4)*

(Pesche). In linea generale, è stata effettuata una selezione tramite *MenuChoices(MenuNo%, ChioiceNo%)*.
Le variabili *Button%, ScRow%* e *ScCol%* potranno essere utilizzate per registrare le azioni a mouse:

```
"mousedown" = MouseGetEvent$ (MenuNo%, ChoiceNo%, Button%,
                              ScRow%, ScCol%)
```

Se viene premuto uno dei due pulsanti del mouse, *MenuGetEvent$( )* agirà esattamente come *INKEY$( )* riportando così qualsiasi tipo di azione di immissione.
All'inizio del capitolo si è già detto come dovrebbe operare un sistema a menu e ora si vedrà come trasferirne l'operatività a *MenuGetEvent$( )*.

```
DO
   IF (viene premuto un tasto) THEN
      MenuGetEvent$ = tasto premuto
      IF (il sistema a menu è attivato) THEN
         IF (viene premuto il tasto ESC) THEN
            IF (viene aperto un menu) THEN
               [chiude il menu e esce dal sistema]
            ELSE
               EXIT FUNCTION
            END IF
         END IF
         IF (viene premuto ALT con la lettera di apertura
             del menu) THEN
            [imposta gli argomenti del menu ed esce dal sistema]
         ELSE
            EXIT FUNCTION
         END IF
      END IF
         EXIT FUNCTION
   END IF

   IF (è stato premuto un pulsante del mouse) THEN
      IF (è stato premuto il pulsante sinistro del mouse) THEN
         IF (il puntatore era all'interno di un menu) THEN
            [imposta gli argomenti del menu e esce dal sistema]
         ELSE
            [imposta gli argomenti del menu e esce dal sistema]
         END IF
      ELSE
         [imposta gli argomenti del mouse e esce dalla funzione]
      END IF
         EXIT FUNCTION
   END IF
LOOP WHILE 1
```

Indubbiamente, quanto sopra riportato è un frammento già abbastanza lungo. Il codice è composto da due elementi primari: il primo controlla se è stato premuto un qualsiasi tasto e ne interpreta la funzione; il secondo controlla la coda del mouse e cerca di interpretare le operazioni avviate a mouse. Si inizierà ora a dare un'occhiata alla parte che leggerà le immissioni da tastiera. Innanzitutto si assegnerà una stringa vuota (" ") a *MenuGetEvent$* e si imposterà un ciclo di tipo *DO ... LOOP WHILE 1* che rimarrà in attesa di un'immissione:

```
FUNCTION MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,
                        ScCol%) STATIC

   DIM InRegs AS RegType, OutRegs AS RegType

   MenuGetEvent$ = ""

   DO
   :
   :
   LOOP WHILE 1
```

Ora si cercherà di leggere la pressione di un tasto. Se il sistema è in attesa di un'immissione ed è disattivata la visualizzazione della barra dei menu (ossia, *Menus OnFlag% = 0*), sarà necessario che il valore venga restituito correttamente:

```
FUNCTION MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,
                        ScCol%) STATIC

   DIM InRegs AS RegType, OutRegs AS RegType

   MenuGetEvent$ = ""

   DO

   InChar$ = INKEY$

   IF InChar$ <> "" THEN
      MenuGetEvent$ = InChar$
      IF MenusOnFlag% = 0 THEN EXIT FUNCTION  'Sistema menu attivo?
      :
      :
   END IF
```

Successivamente, si dovrà controllare se il tasto premuto corrispondeva a *ESC* (che chiude un menu quando questo è aperto). Nel caso in cui sia stato premuto il tasto *ESC* senza che fosse aperto nessun menu, *ESC* verrà trasferito al programma chiamante.

D'altra parte, nel caso in cui fosse stato aperto un menu precedentemente alla pressione di *ESC*, questa operazione risulterà essere un'azione di tipo "*menuclose*"

e verrebbe generata la chiusura del menu impostando il valore restituito in *MenuNo%* per indicare al programma chiamante quale menu è stato chiuso prima di uscire dalla funzione.
Per poter sapere se un menu è stato o meno aperto, si potrebbe impostare un *flag* (*MenusOnFlag%*) per indicare che la barra dei menu è visibile senza, tuttavia, avere la possibilità di controllare quale menu è stato effettivamente aperto. Per questa ragione dovrà essere immessa una nuova variabile (*MenuNowOpen%*, che apparirà abbastanza di frequente in *menuGetEvent$( )*) contenente il numero del menu correntemente aperto. Se il menu risulta aperto essa conterrà il valore 0 (il valore in *MenuNowOpen%* verrà impostato ad ogni apertura o chiusura di un menu). Per controllare se è correntemente aperto un qualsiasi menu sarà quindi necessario controllare il contenuto di *MenuNowOpen%*.

```
FUNCTION MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,
ScCol%) STATIC

   DIM InRegs AS RegType, OutRegs AS RegType

   MenuGetEvent$ = ""

   DO

   InChar$ = INKEY$

   IF InChar$ <> "" THEN
      MenuGetEvent$ = InChar$
      IF MenusOnFlag% = 0 THEN EXIT FUNCTION  'Sistema menu attivo?
      IF ASC(InChar$) = 27 THEN               'Tasto ESCape
         IF MenuNowOpen% = 0 THEN
            EXIT FUNCTION
            :
            :
```

***Nota**: Non è necessario immettere la variabile MenuNowOpen% all'interno di istruzioni COMMON dei menu poiché essa è una semplice variabile locale di MenuGetEvent$( ), a differenza di MenusOnFlag% che, invece, viene anche utilizzata da MenuShow( ) e MenuHide( ).*

Se è stato aperto un menu, la pressione di *ESC* deve generarne la chiusura e poiché le operazioni di apertura e chiusura dei menu saranno abbastanza frequenti sarà necessario creare due sottoprogrammi (*TurnOnMenu( )* e *TurnOffMenu( )*) che siano in grado di visualizzare e nascondere i singoli menu. Queste due routine sono dei semplici adattamenti dei precedenti sottoprogrammi *WindowShow( )* e *WindowHide( )*. Per chiudere un menu precedentemente aperto, per esempio, sarà sufficiente utilizzare un'istruzione del tipo *CALL TurnOffMenu(MenuNowOpen%)*.

```
FUNCTION MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,
                        ScCol%) STATIC

   DIM InRegs AS RegType, OutRegs AS RegType

   MenuGetEvent$ = ""

   DO

   InChar$ = INKEY$

   IF InChar$ <> "" THEN
      MenuGetEvent$ = InChar$
      IF MenusOnFlag% = 0 THEN EXIT FUNCTION  'Sistema menu attivo?
      IF ASC(InChar$) = 27 THEN               'Tasto ESCape
         IF MenuNowOpen% = 0 THEN
            EXIT FUNCTION
      ELSE
         CALL TurnOffMenu(MenuNowOpen%)
         MenuNo% = MenuNowOpen%
         MenuNowOpen% = 0
         MenuGetEvent$ = "menuclose"
         EXIT FUNCTION
      END IF
   END IF                                       'Tasto ESCape
   :
   :
   END IF                                       'Controlla tasto
```

Se il tasto premuto non è stato *ESC* ma la lunghezza della stringa restituita da *INKEY$* è pari a 1, questo significa che è stato premuto un tasto che non ha nessun significato, per cui si ritornerà al programma chiamante. Tuttavia, sarà necessario controllare se è stato premuto il tasto *ALT* (verificandone lo *scan code*, rappresentato dal secondo carattere della stringa restituita da *INKEY$*, al quale viene assegnato il nome *InChar$*). Se questo controllo non verifica la pressione del tasto *ALT*, anche in questo caso significa che è stato premuto un tasto che non ha nessun significato e si ritornerà al programma chiamante.

```
FUNCTION MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,
                        ScCol%) STATIC

   DIM InRegs AS RegType, OutRegs AS RegType

   MenuGetEvent$ = ""

   DO

   InChar$ = INKEY$
```

```
IF InChar$ <> "" THEN
   MenuGetEvent$ = InChar$
   IF MenusOnFlag% = 0 THEN EXIT FUNCTION  'Sistema menu attivo?
   IF ASC(InChar$) = 27 THEN               'Tasto ESCape
      IF MenuNowOpen% = 0 THEN
         EXIT FUNCTION
   ELSE
      CALL TurnOffMenu(MenuNowOpen%)
      MenuNo% = MenuNowOpen%
      MenuNowOpen% = 0
      MenuGetEvent$ = "menuclose"
      EXIT FUNCTION
   END IF
END IF                                         'Tasto ESCape
IF LEN(InChar$) = 1 THEN
   EXIT FUNCTION
ELSE
   ScanCode = ASC(RIGHT$(InChar$,1))
   AltKey$ = ""
   IF ScanCode >= 16 AND ScanCode <= 25 THEN AltKey$⌉
        = MID$("QWERTYUIOP", ScanCode - 15,1)
   IF ScanCode >= 30 AND ScanCode <= 38 THEN AltKey$⌉
        = MID$("ASDFGHJKL", ScanCode - 29,1)
   IF ScanCode >= 44 AND ScanCode <= 50 THEN AltKey$⌉
        = MID$("ZXCVBNM", ScanCode - 43,1)
   IF AltKey$ = "" THEN EXIT FUNCTION
   :
   :
```

A questo punto, se ci si trova ancora all'intero della funzione, si è verificata la pressione di una combinazione di tasti *ALT* e si è inviata la restituzione del valore a *AltKey$*. Sarà ora necessario controllare che la lettera associata ad *ALT* corrisponda alla prima lettera di una selezione all'interno del menu (*sempre che* un menu sia aperto) (figura 4.6). In caso affermativo, si verificherà un'operazione relativa a *menuchoice*. Sarà ora necessario impostare correttamente i valori *MenuNo%* e *ChoiceNo%* in modo che questi possano essere utilizzati dal programma chiamante, chiudere il menu e uscire dal sistema.

```
FUNCTION MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,⌉
                        ScCol%) STATIC

   DIM InRegs AS RegType, OutRegs AS RegType

   MenuGetEvent$ = ""

   DO

   InChar$ = INKEY$
```

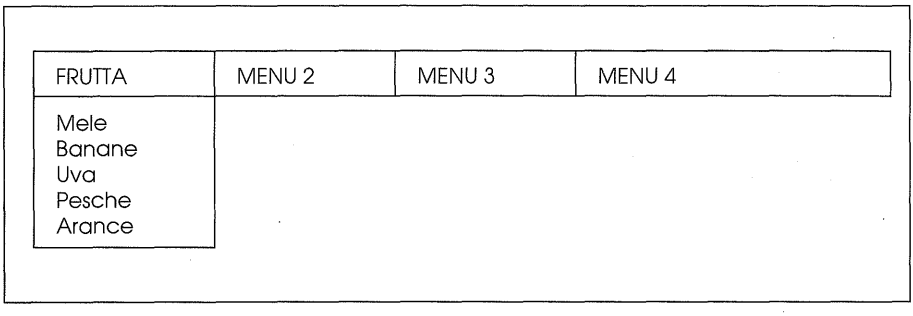

Figura 4.6

```
IF InChar$ <> "" THEN
   MenuGetEvent$ = InChar$
   IF MenusOnFlag% = 0 THEN EXIT FUNCTION  'Sistema menu attivo?
   IF ASC(InChar$) = 27 THEN               'Tasto ESCape
      IF MenuNowOpen% = 0 THEN
         EXIT FUNCTION
   ELSE
      CALL TurnOffMenu(MenuNowOpen%)
      MenuNo% = MenuNowOpen%
      MenuNowOpen% = 0
      MenuGetEvent$ = "menuclose"
      EXIT FUNCTION
   END IF
END IF                                         'Tasto ESCape
IF LEN(InChar$) = 1 THEN
   EXIT FUNCTION
ELSE
   ScanCode = ASC(RIGHT$(InChar$,1))
   AltKey$ = ""
   IF ScanCode >= 16 AND ScanCode <= 25 THEN AltKey$
        = MID$("QWERTYUIOP", ScanCode - 15,1)
   IF ScanCode >= 30 AND ScanCode <= 38 THEN AltKey$
        = MID$("ASDFGHJKL", ScanCode - 29,1)
   IF ScanCode >= 44 AND ScanCode <= 50 THEN AltKey$
        = MID$("ZXCVBNM", ScanCode - 43,1)
   IF AltKey$ = "" THEN EXIT FUNCTION

   IF MenuNowOpen% <> 0 THEN
      FOR i% = 1 TO Rows(MenuNowOpen%)
         IF AltKey$ = UCASE$(MID$(Text(MenuNowOpen%,i%)
                                 ,2,1)) THEN
            MenuNo% = MenuNowOpen%
```

```
            ChoiceNo% = i%
            CALL TurnOffMenu(MenuNowOpen%)
            MenuNowOpen% = 0
            MenuGetEvent$ = "menuchoice"
            EXIT FUNCTION
         END IF
      NEXT i%
   END IF
   :
   :
```

Se anche a questo punto non si sarà usciti ancora dalla funzione, allora la combinazione *ALT-**Tasto*** corrisponde a una voce del menu correntemente aperto. In questo caso sarà necessario controllare e verificare se il menu dovrà essere chiuso (attivando, per esempio, la stessa combinazione *Alt-**Tasto***). In caso di chiusura, verrà avviata un'operazione di tipo *menuclose*.

In caso contrario, si dovrà verificare se la combinazione *ALT-**Tasto*** corrisponde a un diverso nome di menu da aprire. In caso affermativo, verrà avviata un'operazione di tipo *menuopen* che genera l'apertura del menu richiesto (dopo aver precedentemente chiuso il menu eventualmente già aperto). Infine, nel caso in cui la combinazione *Alt-**Tasto*** non ha *nessuna corrispondenza*, il controllo verrà nuovamente trasferito al programma chiamante (tenere presente che *NumMenus%* corrisponde al numero dei menu presenti sulla barra dei menu stessa).

```
FUNCTION MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,
                        ScCol%) STATIC

   DIM InRegs AS RegType, OutRegs AS RegType

   MenuGetEvent$ = ""

   DO

   InChar$ = INKEY$

   IF InChar$ <> "" THEN
      MenuGetEvent$ = InChar$
      IF MenusOnFlag% = 0 THEN EXIT FUNCTION  'Sistema menu attivo?
      IF ASC(InChar$) = 27 THEN               'Tasto ESCape
         IF MenuNowOpen% = 0 THEN
            EXIT FUNCTION
      ELSE
         CALL TurnOffMenu(MenuNowOpen%)
         MenuNo% = MenuNowOpen%
         MenuNowOpen% = 0
         MenuGetEvent$ = "menuclose"
         EXIT FUNCTION
      END IF
```

```
END IF                                          'Tasto ESCape
IF LEN(InChar$) = 1 THEN
   EXIT FUNCTION
ELSE
   ScanCode = ASC(RIGHT$(InChar$,1))
   AltKey$ = ""
   IF ScanCode >= 16 AND ScanCode <= 25 THEN AltKey$⌉
        = MID$("QWERTYUIOP", ScanCode - 15,1)
   IF ScanCode >= 30 AND ScanCode <= 38 THEN AltKey$⌉
        = MID$("ASDFGHJKL", ScanCode - 29,1)
   IF ScanCode >= 44 AND ScanCode <= 50 THEN AltKey$⌉
        = MID$("ZXCVBNM", ScanCode - 43,1)
   IF AltKey$ = "" THEN EXIT FUNCTION

   IF MenuNowOpen% <> 0 THEN
      FOR i% = 1 TO Rows(MenuNowOpen%)
         IF AltKey$ = UCASE$(MID$(Text(MenuNowOpen%,i%)⌉
                                  ,2,1)) THEN
            MenuNo% = MenuNowOpen%
            ChoiceNo% = i%
            CALL TurnOffMenu(MenuNowOpen%)
            MenuNowOpen% = 0
            MenuGetEvent$ = "menuchoice"
            EXIT FUNCTION
         END IF
      NEXT i%
   END IF
   FOR i% = 1 TO NumMenus%
      IF AltKey$ = UCASE$(MID$(Bar$,10*(i%-1)+2,1)) THEN
         IF MenuNowOpen% = i% THEN
            MenuNo% = i%
            CALL TurnOffMenu(i%)
            MenuNowOpen% = 0
            MenuGetEvent$ = "menuclose"
            EXIT FUNCTION
         END IF
         IF MenuNowOpen% <> 0 THEN CALL TurnOffMenu⌉
                 (MenuNowOpen%)
         MenuNo% = i%
         ChoiceNo% = 0
         MenuNowOpen% = i%
         CALL TurnOnMenu(i%)
         MenuGetEvent$ = "menuopen"
      END IF
   NEXT i%                              'For i% = 1 TO NumMenus%
END IF                                  'Controlla lunghezza char
   EXIT FUNCTION
END IF                                  'Controlla tasto
```

E con questo si è terminata la routine per la lettura di parte di *MenuGetEvent$( )*, tenendo presente, tuttavia, che si è presa in esame solo metà della funzione globale *MenuGetEvent$( )*. Si passerà ora ad analizzare la parte che controlla la gestione delle operazioni del mouse.

Di seguito si lavorerà unicamente sulla parte del codice che tratterà del pulsante sinistro, quello che risulterà operativo quando si intende lavorare con un sistema a menu. La parte della funzione *MenuGetEvent$( )* che gestisce il pulsante destro del mouse è relativamente semplice e verrà presa in esame successivamente poiché la sua definizione non richiede una trattazione molto lunga.

La parte che prende in esame il pulsante sinistro del mouse potrà essere suddivisa in due frammenti che controllano la pressione e il rilascio del pulsante stesso. Innanzitutto sarà necessario controllarne la pressione inviando la posizione del puntatore (espressa in righe e colonne), rispettivamente, alle variabili *ScRow%* e *ScCol%* perché queste siano poi in grado di restituirne il valore al programma chiamante (e perché queste variabili verranno anche utilizzate in altra parte del programma). Per eseguire questa operazione sarà sufficiente indirizzare i valori restituiti ai registri *dx* e *cx*, effettuare una divisione intera per 8 e aggiungervi 1.

Si noti che se *ScRow%* era pari a *1* e la barra dei menu era visualizzata (*MenusOnFlag% = 1*), allora si verificherà un'operazione di tipo *menuopen* (sempre che il puntatore del mouse sia stato precedentemente posizionato sul nome di un menu). In questo caso il valore restituito verrà inviato a *MenuNo%*, verrà aperto il menu richiesto e si uscirà. In caso contrario, verrà comunque memorizzata in *NumberTimes%* (contenente il numero di volte che è stato premuto il pulsante sinistro del mouse) l'operazione a mouse avviata.

```
REM *** Controlla la pressione del pulsante sinistro

Button% = 0
InRegs.bx = 0
InRegs.ax = 5
CALL INTERRUPT(&H33, InRegs, OutRegs)

NumberTimes% = OutRegs.bx  'Numero pressioni pulsante sinistro
ScRow% = OutRegs.dx \ 8 + 1
ScCol% = OutRegs.cx \ 8 + 1

IF NumberTimes% <> 0 THEN
   IF MenusOnFlag% AND (ScRow% = 1) THEN
      MenubarCandidate% = ScCol% \ 10 + 1
      IF MenubarCandidate% <= NumMenus% THEN
         MenuNo% = MenubarCandidate%
         ChoiceNo% = 0
         MenuNowOpen% = MenubarCandidate%
         CALL TurnOnMenu(MenubarCandidate%)
         MenuGetEvent$ = "menuopen"
         EXIT FUNCTION
```

```
      END IF
   ELSE
      MenuGetEvent$ = "mousedown"
      EXIT FUNCTION
   END IF
END IF
:
:
```

Quanto sopra definisce e conclude quindi la gestione della pressione del pulsante sinistro del mouse (poiché, in effetti, la selezione della voce di menu verrà confermata solo al momento del rilascio del pulsante). Si analizzerà ora la sezione del codice che prende in esame la gestione del rilascio del pulsante. Ancora una volta, quando verrà rilasciato il pulsante del mouse, la posizione del puntatore verrà registrata in *ScRow%* e *ScCol%*.

La procedura, questa volta, è un po' più complessa; se la barra dei menu è visibile (*MenusOnFlag% = 1*) ed è stato aperto un qualsiasi menu (*MenuNowOpen%* diverso da 0) sarà necessario controllare se è stata avviata un'operazione su menu. In caso contrario si uscirà dalla funzione e l'operazione di rilascio del pulsante verrà registrata per un futuro controllo. Di seguito viene riportato il frammento di codice che prende in esame il rilascio del pulsante del mouse.

```
REM *** Controlla il rilascio del pulsante sinistro

InRegs.bx = 0
InRegs.ax = 6
CALL INTERRUPT(&H33, InRegs, OutRegs)

NumberTimes% = OutRegs.bx
ScRow% = OutRegs.dx \ 8 + 1
ScCol% = OutRegs.cx \ 8 + 1

IF NumberTimes% <> 0 THEN
   IF MenusOnFlag% AND (MenuNowOpen% <> 0) THEN
      :
      [Controlla l'operazione sui menu]
      :
   ELSE
      MenuGetEvent$ = "mouseup"
      EXIT FUNCTION
   END IF 'Menu open
END IF 'NumberTimes% <> 0
```

Per poter controllare un'operazione sui menu sarà necessario controllare se il puntatore del mouse era posizionato all'interno di un menu dopo che questo è stato aperto quando si è rilasciato il pulsante sinistro. Nel caso in cui il puntatore del mouse fosse all'esterno della finestra contenente le opzioni di menu (posizionato quindi in una colonna e riga al di fuori dell'intervallo definito) si presupporrà la chiusura del

menu precedentemente aperto, avviando così un'operazione di tipo *menuclose*. Di seguito viene presentato il frammento di codice che definisce questa eventualità.

```
REM *** Controlla il rilascio del pulsante sinistro

InRegs.bx = 0
InRegs.ax = 6
CALL INTERRUPT(&H33, InRegs, OutRegs)

NumberTimes% = OutRegs.bx
ScRow% = OutRegs.dx \ 8 + 1
ScCol% = OutRegs.cx \ 8 + 1

IF NumberTimes% <> 0 THEN
   IF MenusOnFlag% AND (MenuNowOpen% <> 0) THEN
      IF ScRow% >= 2 AND ScRow% <= 2 + Rows(MenuNowOpen%) THEN
         IF ScCol% >= 1 + 10 * (MenuNowOpen%-1) AND ScCol%
                                <= 10*(MenuNowOpen%) THEN
         :
         [Il puntatore era in un menu precedentemente aperto
         quando è stato rilasciato il pulsante sinistro: viene
         selezionata un'opzione dal menu]
         :
         ELSE
            MenuNo% = MenuNowOpen%
            CALL TurnOffMenu(MenuNowOpen%)
            MenuNowOpen% = 0
            MenuGetEvent$ = "menuclose"
            EXIT FUNCTION
         END IF
      ELSE
         MenuNo% = MenuNowOpen%
         CALL TurnOffMenu(MenuNowOpen%)
         MenuNowOpen% = 0
         MenuGetEvent$ = "menuclose"
         EXIT FUNCTION
      END IF
   ELSE
      MenuGetEvent$ = "mouseup"
      EXIT FUNCTION
   END IF 'Menu open
END IF 'NumberTimes% <> 0
```

In effetti, il frammento di codice sopra riportato è il "cuore" della gestione delle operazioni a mouse sui menu poiché si presuppone che il pulsante sia stato rilasciato quando il puntatore era posizionato all'interno di un menu precedentemente aperto. Sarà ora tuttavia necessario verificare quale selezione verrà attivata al rilascio del pulsante. Poiché la prima riga di ciascun menu è sempre la seconda riga dello

schermo, la selezione effettuata (*ChoiceNo%*) sarà sempre pari a *ScRow% – 1* (si veda la figura 4.7). La selezione effettuata verrà memorizzata e il menu verrà successivamente chiuso.

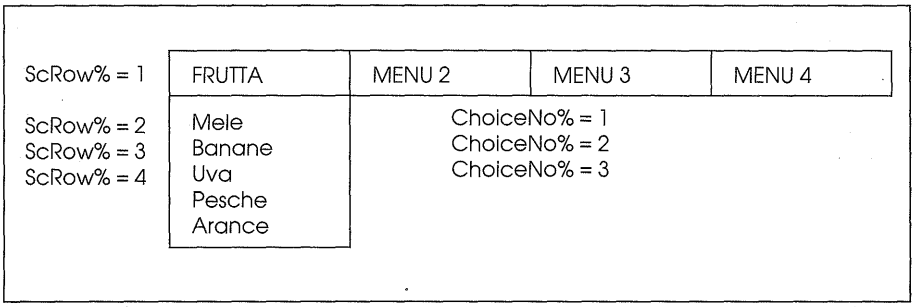

Figura 4.7

```
REM *** Controlla il rilascio del pulsante sinistro

InRegs.bx = 0
InRegs.ax = 6
CALL INTERRUPT(&H33, InRegs, OutRegs)

NumberTimes% = OutRegs.bx
ScRow% = OutRegs.dx \ 8 + 1
ScCol% = OutRegs.cx \ 8 + 1

IF NumberTimes% <> 0 THEN
   IF MenusOnFlag% AND (MenuNowOpen% <> 0) THEN
      IF ScRow% >= 2 AND ScRow% <= 2 + Rows(MenuNowOpen%) THEN
         IF ScCol% >= 1 + 10 * (MenuNowOpen%-1) AND ScCol%
                  <= 10*(MenuNowOpen%) THEN
            ChoiceNo% = ScRow% - 1
            CALL TurnOffMenu(MenuNowOpen%)
            MenuNo% = MenuNowOpen%
            MenuNowOpen% = 0
            MenuGetEvent$ = "menuchoice"
            EXIT FUNCTION
         ELSE
            MenuNo% = MenuNowOpen%
            CALL TurnOffMenu(MenuNowOpen%)
            MenuNowOpen% = 0
            MenuGetEvent$ = "menuclose"
            EXIT FUNCTION
         END IF
      ELSE
```

```
            MenuNo% = MenuNowOpen%
            CALL TurnOffMenu(MenuNowOpen%)
            MenuNowOpen% = 0
            MenuGetEvent$ = "menuclose"
            EXIT FUNCTION
         END IF
      ELSE
         MenuGetEvent$ = "mouseup"
         EXIT FUNCTION
      END IF 'Menu open
   END IF 'NumberTimes% <> 0
```

E con questo si è conclusa la parte relativa al mouse di *MenuGetEvent$( )* (relativamente al solo pulsante sinistro del mouse). La parte che tratta del pulsante destro è molto più semplice da gestire poiché essa prende in esame unicamente azioni di tipo *mousedown* e *mouseup* lasciando la sua trattazione a quei programmatori che ne abbiano un'effettiva necessità. Il listato 4.4 riporta l'intera funzione.

```
DECLARE FUNCTION MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%,
                                ScRow%, ScCol%)
DECLARE SUB TurnOnMenu (MNumber%)
DECLARE SUB TurnOffMenu (MNumber%)

TYPE RegType
   ax      AS INTEGER
   bx      AS INTEGER
   cx      AS INTEGER
   dx      AS INTEGER
   bp      AS INTEGER
   si      AS INTEGER
   di      AS INTEGER
   flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

   CONST MaxMenus = 7
   DIM Rows(1 TO MaxMenus) AS INTEGER
   DIM Cols(1 TO MaxMenus) AS INTEGER
   DIM TopRow(1 TO MaxMenus) AS INTEGER
   DIM TopCol(1 TO MaxMenus) AS INTEGER
   DIM BotRow(1 TO MaxMenus) AS INTEGER
   DIM BotCol(1 TO MaxMenus) AS INTEGER
   DIM Attribute(1 TO MaxMenus) AS INTEGER
```

continua

Listato 4.4 *La funzione MenuGetEvent$ per leggere le azioni avviate dai menu*

```
    DIM Text(1 TO MaxMenus, 25) AS STRING
    DIM OldText(1 TO MaxMenus, 25) AS STRING
    DIM OldAttrb(1 TO MaxMenus, 25) AS STRING
    COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,⌉
         TopRow() AS INTEGER, TopCol() AS INTEGER, BotRow()⌉
         AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /MenuB/ Attribute() AS INTEGER, Text()⌉
         AS STRING, OldText() AS STRING, OldAttrb() AS STRING
    COMMON SHARED /MenuC/ Bar$, BarAttrb, MenusOnFlag%,⌉
         NumMenus%, OldBar$, OldBarAttrb$

FUNCTION MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,⌉
         ScCol%) STATIC

    DIM InRegs AS RegType, OutRegs AS RegType

    MenuGetEvent$ = ""

    DO
    InChar$ = INKEY$

    IF InChar$ <> "" THEN
       MenuGetEvent$ = InChar$
       IF MenusOnFlag% = 0 THEN EXIT FUNCTION
                                          'Sistema menu attivo?
       IF ASC(InChar$) = 27 THEN          'Tasto ESCape
          IF MenuNowOpen% = 0 THEN
             EXIT FUNCTION
       ELSE
          CALL TurnOffMenu(MenuNowOpen%)
          MenuNo% = MenuNowOpen%
          MenuNowOpen% = 0
          MenuGetEvent$ = "menuclose"
          EXIT FUNCTION
       END IF
    END IF                                          'Tasto ESCape
    IF LEN(InChar$) = 1 THEN
       EXIT FUNCTION
    ELSE
       ScanCode = ASC(RIGHT$(InChar$,1))
       AltKey$ = ""
       IF ScanCode >= 16 AND ScanCode <= 25 THEN AltKey$⌉
               = MID$("QWERTYUIOP", ScanCode - 15,1)
       IF ScanCode >= 30 AND ScanCode <= 38 THEN AltKey$⌉
               = MID$("ASDFGHJKL", ScanCode - 29,1)
```

continua

Listato 4.4 *La funzione MenuGetEvent$ per leggere le azioni avviate dai menu*

```
    IF ScanCode >= 44 AND ScanCode <= 50 THEN AltKey$
            = MID$("ZXCVBNM", ScanCode - 43,1)
    IF AltKey$ = "" THEN EXIT FUNCTION

    IF MenuNowOpen% <> 0 THEN
       FOR i% = 1 TO Rows(MenuNowOpen%)
          IF AltKey$ = UCASE$(MID$(Text(MenuNowOpen%,i%),2,1))
                     THEN
             MenuNo% = MenuNowOpen%
             ChoiceNo% = i%
             CALL TurnOffMenu(MenuNowOpen%)
             MenuNowOpen% = 0
             MenuGetEvent$ = "menuchoice"
             EXIT FUNCTION
          END IF
       NEXT i%
    END IF

    FOR i% = 1 TO NumMenus%
       IF AltKey$ = UCASE$(MID$(Bar$,10*(i%-1)+2,1)) THEN
          IF MenuNowOpen% = i% THEN
             MenuNo% = i%
             CALL TurnOffMenu(i%)
             MenuNowOpen% = 0
             MenuGetEvent$ = "menuclose"
             EXIT FUNCTION
          END IF
          IF MenuNowOpen% <> 0 THEN CALL TurnOffMenu
                 (MenuNowOpen%)
          MenuNo% = i%
          ChoiceNo% = 0
          MenuNowOpen% = i%
          CALL TurnOnMenu(i%)
          MenuGetEvent$ = "menuopen"
       END IF
    NEXT i%                              'For i% = 1 TO NumMenus%
 END IF                                  'Controlla lunghezza char
    EXIT FUNCTION
 END IF                                  ' Controlla tasto


 REM *** Controlla la pressione del pulsante sinistro

 Button% = 0
 InRegs.bx = 0
```

continua

Listato 4.4 *La funzione MenuGetEvent$ per leggere le azioni avviate dai menu*

```
InRegs.ax = 5
CALL INTERRUPT(&H33, InRegs, OutRegs)

NumberTimes% = OutRegs.bx 'Numero pressioni pulsante sinistro
ScRow% = OutRegs.dx \ 8 + 1
ScCol% = OutRegs.cx \ 8 + 1

IF NumberTimes% <> 0 THEN
   IF MenusOnFlag% AND (ScRow% = 1) THEN
      MenubarCandidate% = ScCol% \ 10 + 1
      IF MenubarCandidate% <= NumMenus% THEN
         MenuNo% = MenubarCandidate%
         ChoiceNo% = 0
         MenuNowOpen% = MenubarCandidate%
         CALL TurnOnMenu(MenubarCandidate%)
         MenuGetEvent$ = "menuopen"
         EXIT FUNCTION
      END IF
   ELSE
      MenuGetEvent$ = "mousedown"
      EXIT FUNCTION
   END IF
END IF


REM *** Controlla il rilascio del pulsante sinistro

InRegs.bx = 0
InRegs.ax = 6

CALL INTERRUPT(&H33, InRegs, OutRegs)

NumberTimes% = OutRegs.bx
ScRow% = OutRegs.dx \ 8 + 1
ScCol% = OutRegs.cx \ 8 + 1

IF NumberTimes% <> 0 THEN
   IF MenusOnFlag% AND (MenuNowOpen% <> 0) THEN
      IF ScRow% >= 2 AND ScRow% <= 2 + Rows(MenuNowOpen%) THEN
         IF ScCol% >= 1 + 10 * (MenuNowOpen%-1) AND ScCol%
                        <= 10*(MenuNowOpen%) THEN
            ChoiceNo% = ScRow% - 1
            CALL TurnOffMenu(MenuNowOpen%)
            MenuNo% = MenuNowOpen%
            MenuNowOpen% = 0
```

continua

Listato 4.4 *La funzione MenuGetEvent$ per leggere le azioni avviate dai menu*

```
            MenuGetEvent$ = "menuchoice"
            EXIT FUNCTION
         ELSE
            MenuNo% = MenuNowOpen%
            CALL TurnOffMenu(MenuNowOpen%)
            MenuNowOpen% = 0
            MenuGetEvent$ = "menuclose"
            EXIT FUNCTION
         END IF
      ELSE
         MenuNo% = MenuNowOpen%
         CALL TurnOffMenu(MenuNowOpen%)
         MenuNowOpen% = 0
         MenuGetEvent$ = "menuclose"
         EXIT FUNCTION
      END IF
   ELSE
      MenuGetEvent$ = "mouseup"
      EXIT FUNCTION
   END IF 'Menu open
END IF 'NumberTimes% <> 0

REM *** Controlla la pressione del pulsante destro

Button% = 1
InRegs.bx = 1
InRegs.ax = 5
CALL INTERRUPT(&H33, InRegs, OutRegs)

NumberTimes% = OutRegs.bx
ScRow% = OutRegs.dx \ 8 + 1
ScCol% = OutRegs.cx \ 8 + 1

IF NumberTimes% <> 0 THEN
   MenuGetEvent$ = "mousedown"
   EXIT FUNCTION
END IF

REM *** Controlla il rilascio del pulsante destro

InRegs.bx = 1
InRegs.ax = 6
CALL INTERRUPT(&H33, InRegs, OutRegs)
NumberTimes% = OutRegs.bx
ScRow% = OutRegs.dx \ 8 + 1
```

continua

Listato 4.4 *La funzione MenuGetEvent$ per leggere le azioni avviate dai menu*

```
ScCol% = OutRegs.cx \ 8 + 1

IF NumberTimes% <> 0 THEN
   MenuGetEvent$ = "mousedown"
   EXIT FUNCTION
END IF

LOOP WHILE 1

END FUNCTION


SUB TurnOnMenu (MNumber%)

   DIM InRegs AS RegType, OutRegs AS RegType

   CurRow% = CSRLIN
   CurCol% = POS(0)

   InRegs.ax = 2
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   REM Salva la precedente sezione dello schermo

   rr = TopRow(MNumber%)
   cc = TopCol(MNumber%)
   rt = Rows(MNumber%)
   ct = Cols(MNumber%)

   FOR i = 1 TO rt
      temp1$ = ""
      temp2$ = ""
      FOR j = 1 TO ct
         temp1$ = temp1$ + CHR$(SCREEN(rr + i - 1, cc + j - 1))
         temp2$ = temp2$ + CHR$(SCREEN(rr + i - 1, cc + j
                                    - 1, 1))
      NEXT j
      OldText(MNumber%, i) = temp1$
      OldAttrb(MNumber%, i) = temp2$
   NEXT i

   REM Visualizza ora il nuovo testo
   TR = TopRow(MNumber%)
   TC = TopCol(MNumber%)
   LOCATE TR, TC
```

continua

Listato 4.4 *La funzione MenuGetEvent$ per leggere le azioni avviate dai menu*

```
    InRegs.cx = 1

    FOR i = 1 TO Rows(MNumber%)
       OR$ = Text(MNumber%, i)
       FOR j = 1 TO Cols(MNumber%)
          InRegs.ax = &H900 + ASC(MID$(OR$, j, 1))
          InRegs.bx = Attribute(MNumber%)
          CALL INTERRUPT(&H10, InRegs, OutRegs)
          LOCATE TR + i - 1, TC + j
       NEXT j
          LOCATE TR + i, TC
    NEXT i

    InRegs.ax = 1
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    LOCATE CurRow%, CurCol%

 END SUB


 SUB TurnOffMenu (MNumber%)

    CurRow% = CSRLIN
    CurCol% = POS(0)

    DIM InRegs AS RegType, OutRegs AS RegType

    InRegs.ax = 2
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    REM Stampa il testo precedente

    TR = TopRow(MNumber%)
    TC = TopCol(MNumber%)
    LOCATE TR, TC
    InRegs.cx = 1

    FOR i = 1 TO Rows(MNumber%)
       OR$ = OldText(MNumber%, i)
       OC$ = OldAttrb(MNumber%, i)
       FOR j = 1 TO Cols(MNumber%)
          InRegs.ax = &H900 + ASC(MID$(OR$, j, 1))
          InRegs.bx = ASC(MID$(OC$, j, 1))
          CALL INTERRUPT(&H10, InRegs, OutRegs)
```

continua

Listato 4.4 *La funzione MenuGetEvent$ per leggere le azioni avviate dal menu*

```
            LOCATE TR + i - 1, TC + j
         NEXT j
            LOCATE TR + i, TC
      NEXT i

      InRegs.ax = 1
      CALL INTERRUPT(&H33, InRegs, OutRegs)

      LOCATE CurRow%, CurCol%

END SUB
```

Listato 4.4 *La funzione MenuGetEvent$ per leggere le azioni avviate dai menu*

Come si noterà, la funzione ha dimensioni davvero estese e notare che essa contiene due sottoprogrammi, *TurnOnMenu( )* e *TurnOffMenu( )*.

PROVA DI ATTESA DI IMMISSIONI DAI MENU

Di seguito viene riportato come utilizzare *MenuGetEvent$( )* in modo che questa sia in grado di leggere immissioni da tastiera, da mouse o da menu.

```
DECLARE FUNCTION MenuInitialize% (MenuNames( ) AS STRING,
                 MenuChoices( ) AS STRING, BarAttrb%,
                 MenuAttrbs( ) AS INTEGER)
DECLARE SUB MenuShow( )
DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE FUNCTION MouseGetEvent$(MenuNo%, ChoiceNo%, Button%,
                                Row%, Col%)

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
   MenuChoices(2, 1) = "Piselli"
   MenuChoices(2, 2) = "Grano"
   MenuChoices(2, 3) = "Broccoli"
```

```
MenuNames(3) = "Carne"
MenuChoices(3, 1) = "Pollo"
MenuChoices(3, 2) = "Maiale"
MenuChoices(3, 3) = "Manzo"
MenuChoices(3, 4) = "Pesce"

FOR i= 1 TO 3
   MenuAttrbs(i) = &H61
NEXT i
BarAttrb% = &H24

Check% = MenuInitialize%(MenuNames( ), MenuChoices( ),
         BarAttrb%, MenuAttrbs( ))
IF Check% = 0 THEN
   PRINT "ERRORE!"
ELSE
   PRINT "Menu impostato"
END IF

CALL MenuShow

Check% = MouseInitialize%

IF Check% = 0 THEN PRINT "Mouse non attivato"

CALL MouseShowCursor

PRINT "Selezionare una voce di menu."
DO
Check$ = MenuGetEvent$(MenuNo%, ChoiceNo%, Button%, Row%, Col%)
LOOP UNTIL Check$ = "menuchoice"

PRINT "La selezione effettuata è stata ",MenuChoices
      (MenuNo%, ChoiceNo%)
```

Il programma di esempio sopra riportato richiede qualche breve spiegazione. Ormai si dovrebbero conoscere le procedure per impostare i menu utilizzando *MenuInitialize%( )* e come richiamare direttamente *MenuShow( )* immediatamente dopo questa operazione. Tuttavia, si dovrebbe notare che il sistema a menu *non attiva* automaticamente l'operatività del mouse (poiché il suo uso non è sempre espressamente richiesto). Per questa ragione, l'uso del mouse dovrà essere dichiarato con *MouseInitialize%( )* e *MouseShowCursor( )* e solo dopo queste operazioni il mouse risulterà realmente attivo per il sistema a menu.

```
:
:
CALL MenuShow
```

```
Check% = MouseInitialize%

IF Check% = 0 THEN PRINT "Mouse non attivato."

CALL MouseShowCursor
```

Di seguito viene riportato come il programma imposta tutte le variabili necessarie.

```
CALL MenuShow

Check% = MouseInitialize%

IF Check% = 0 THEN PRINT "Mouse non attivato"

CALL MouseShowCursor

PRINT "Selezionare una voce di menu."
DO
Check$ = MenuGetEvent$(MenuNo%, ChoiceNo%, Button%, Row%, Col%)
LOOP UNTIL Check$ = "menuchoice"

PRINT "La selezione effettuata è stata ",MenuChoices]
      (MenuNo%, ChoiceNo%)
```

A questo punto sarà necessario richiamare *menuGetEvent$( )* e mantenere attiva l'invocazione fintanto che non verrà restituito un valore che definisce un'azione di tipo *menuchoice* che garantirà che la selezione effettuata è stata una selezione *ChoiceNo%* del menu *menuNo%*. Inoltre si saprà quale selezione è stata effettuata poiché il sistema a menu è stato inizializzato dall'utente. In effetti la selezione effettuata corrisponde a *menuChoices(MenuNo%, ChoiceNo%)*.

Quando questo programma verrà avviato, verrà visualizzata sullo schermo una barra dei menu che rimarrà in attesa di una selezione. Una volta effettuata la selezione (servendosi del mouse o della tastiera) verrà visualizzata la stringa corrispondente alla selezione effettuata (*Pesche, Mele,* ecc.).

La seconda funzione di immissione sulla quale si dovrà ora lavorare è *menuCheckEvent$( )* che differisce da *MenuGetEvent$( )* in quanto il sistema non rimane in attesa di un'immissione.

LETTURA DI IMMISSIONE DA MENU SENZA ATTESA

Si tratta, in sostanza dell'*INKEY$* del sistema a menu. La funzione che si creerà è uguale alla precedente funzione *MenuGetEvent$( )* con la sola differenza che *Menu-*

CheckEvent$( ) non rimane in attesa di un'azione. Se non vi è alcuna azione a mouse, a tastiera o da menu in attesa, *MenuCheckEvent$( )* viene impostata a ""e la funzione inizia nuovamente il suo ciclo.
D'altra parte, se vi è qualcosa nella coda delle operazioni a mouse, o se vi è un tasto in attesa, la funzione *menuCheckEvent$( )* agisce esattamente come *MenuGetEvent$( )*. La sola differenza riguarda i valori restituiti: *MenuCheckEvent$( )* è in grado di restituire una stringa vuota ("") mentre *MenuGetEvent$( )* non è mai in grado di generare questo tipo di restituzione. Di seguito viene indicato come utilizzare la funzione:

```
InString$ = MenuCheckEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,⌉
                            ScCol%)
```

Dopo questa riga, *InString$* verrà impostata a uno dei seguenti valori:

InString$	*Descrizione*
" " (stringa vuota)	Nessun tasto, azione a mouse o azione da menu in attesa.
STRING di lunghezza 1	Pressione di un singolo carattere. La lettura di questa immissione è identica a quella che viene eseguita da *INKEY$*.
STRING di lunghezza 2	Pressione di un tasto con codice ASCII di tipo esteso. La lettura di questa immissione è identica a quella che viene eseguita da *INKEY$*.
"mousedown"	Indica quale pulsante del mouse è stato premuto: *Button% = 0* indica il pulsante sinistro, *Button% = 1* indica il pulsante destro. Le coordinate video (*ScRow%* e *ScCol%*) vengono espresse con un valore di righe e colonne video (*1-25* e *1-80*).
"mouseup"	Indica quale pulsante del mouse è stato rilasciato: *Button% = 0* indica il pulsante sinistro, *Button% = 1* indica il pulsante destro. Le coordinate video (*ScRow%* e *ScCol%*) vengono espresse con un valore di righe e colonne video (*1-25* e *1-80*).
"menuopen"	L'utente ha aperto un menu. Il numero del menu viene immesso in *MenuNo%*.
"menuclose"	L'utente ha chiuso un menu. Il numero del menu viene immesso in *MenuNo%*.
"menuchoice"	Indica la selezione di menu. *MenuNo% = numero del menu*; *ChoiceNo% = numero dell'opzione del menu*. I numeri dei menu e delle relative opzioni corrispondono agli indici degli array che sono stati trasferiti a *MenuInitialize%( )*.

La funzione è pressoché identica a *MenuGetEvent$( )* che è precedentemente stata presentata. Vi sono solo due differenze sostanziali: la prima è che il ciclo *DO ... LOOP WHILE 1* di *MenuGetEvent$( )* è stato rimosso. Come si ricorda, il ciclo appariva come quello di seguito riportato:

```
DO
   IF (viene premuto un tasto) THEN
      MenuGetEvent$ = tasto premuto
      IF (il sistema a menu è attivato) THEN
         IF (viene premuto il tasto ESC) THEN
            IF (viene aperto un menu) THEN
               [chiude il menu e esce dal sistema]
            ELSE
               EXIT FUNCTION
            END IF
         END IF
         IF (viene premuto ALT con la lettera di apertura
            del menu) THEN
            [imposta gli argomenti del menu ed esce dal sistema]
         ELSE
            EXIT FUNCTION
         END IF
      END IF
         EXIT FUNCTION
   END IF

   IF (è stato premuto un pulsante del mouse) THEN
      IF (è stato premuto il pulsante sinistro del mouse) THEN
         IF (il puntatore era all'interno di un menu) THEN
            [imposta gli argomenti del menu e esce dal sistema]
         ELSE
            [imposta gli argomenti del menu e esce dal sistema]
         END IF
      ELSE
         [imposta gli argomenti del mouse e esce dalla funzione]
      END IF
         EXIT FUNCTION
   END IF
LOOP WHILE 1
```

Senza questo ciclo, la funzione non rimarrà più in attesa di un'immissione. La seconda differenza rispetto alla funzione precedente, sta nel fatto che ogni occasione in cui appariva *MenuGetEvent* è stata sostituita da *MenuCheckEvent*. Il listato 4.5 riporta l'intera funzione.

```
DECLARE FUNCTION MenuCheckEvent$ (MenuNo%, ChoiceNo%, Button%,
                                  ScRow%, ScCol%)
DECLARE SUB TurnOnMenu (MNumber%)
DECLARE SUB TurnOffMenu (MNumber%)
```

continua

Listato 4.5 *La funzione MenuCheckEvent$ per leggere le azioni senza attesa*

```
TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,⌉
                       OutRegs AS RegType)

    CONST MaxMenus = 7
    DIM Rows(1 TO MaxMenus) AS INTEGER
    DIM Cols(1 TO MaxMenus) AS INTEGER
    DIM TopRow(1 TO MaxMenus) AS INTEGER
    DIM TopCol(1 TO MaxMenus) AS INTEGER
    DIM BotRow(1 TO MaxMenus) AS INTEGER
    DIM BotCol(1 TO MaxMenus) AS INTEGER
    DIM Attribute(1 TO MaxMenus) AS INTEGER
    DIM Text(1 TO MaxMenus, 25) AS STRING
    DIM OldText(1 TO MaxMenus, 25) AS STRING
    DIM OldAttrb(1 TO MaxMenus, 25) AS STRING

    COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,⌉
         TopRow() AS INTEGER, TopCol() AS INTEGER, BotRow()⌉
         AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /MenuB/ Attribute() AS INTEGER, Text()⌉
         AS STRING, OldText() AS STRING, OldAttrb() AS STRING
    COMMON SHARED /MenuC/ Bar$, BarAttrb, MenusOnFlag%,⌉
         NumMenus%, OldBar$, OldBarAttrb$

FUNCTION MenuCheckEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,⌉
                          ScCol%) STATIC

    DIM InRegs AS RegType, OutRegs AS RegType

    MenuCheckEvent$ = ""

    InChar$ = INKEY$

    IF InChar$ <> "" THEN
       MenuCheckEvent$ = InChar$
```

continua

Listato 4.5 *La funzione MenuCheckEvent$ per leggere le azioni senza attesa*

```
IF MenusOnFlag% = 0 THEN EXIT FUNCTION
                                        'Sistema menu attivo?
IF ASC(InChar$) = 27 THEN               'Tasto ESCape
   IF MenuNowOpen% = 0 THEN
      EXIT FUNCTION
   ELSE
      CALL TurnOffMenu(MenuNowOpen%)
      MenuNo% = MenuNowOpen%
      MenuNowOpen% = 0
      MenuCheckEvent$ = "menuclose"
      EXIT FUNCTION
   END IF
END IF          'Escape key
IF LEN(InChar$) = 1 THEN
   EXIT FUNCTION
ELSE
   ScanCode = ASC(RIGHT$(InChar$, 1))
   AltKey$ = ""
   IF ScanCode >= 16 AND ScanCode <= 25 THEN AltKey$
        = MID$("QWERTYUIOP", ScanCode - 15, 1)
   IF ScanCode >= 30 AND ScanCode <= 38 THEN AltKey$
        = MID$("ASDFGHJKL", ScanCode - 29, 1)
   IF ScanCode >= 44 AND ScanCode <= 50 THEN AltKey$
        = MID$("ZXCVBNM", ScanCode - 43, 1)
   IF AltKey$ = "" THEN EXIT FUNCTION

   IF MenuNowOpen% <> 0 THEN
      FOR I% = 1 TO Rows(MenuNowOpen%)
         IF AltKey$ = UCASE$(MID$(Text(MenuNowOpen%,
                        I%), 2, 1)) THEN
            MenuNo% = MenuNowOpen%
            ChoiceNo% = I%
            CALL TurnOffMenu(MenuNowOpen%)
            MenuNowOpen% = 0
            MenuCheckEvent$ = "menuchoice"
            EXIT FUNCTION
         END IF
      NEXT I%
   END IF

   FOR I% = 1 TO NumMenus%

      IF AltKey$ = UCASE$(MID$(Bar$, 10 * (I% - 1)
                     + 2, 1)) THEN
         IF MenuNowOpen% = I% THEN
```

continua

Listato 4.5 *La funzione MenuCheckEvent$ per leggere le azioni senza attesa*

```
                MenuNo% = I%
                CALL TurnOffMenu(I%)
                MenuNowOpen% = 0
                MenuCheckEvent$ = "menuclose"
                EXIT FUNCTION
             END IF
             IF MenuNowOpen% <> 0 THEN CALL TurnOffMenu
                     (MenuNowOpen%)
             MenuNo% = I%
             ChoiceNo% = 0
             MenuNowOpen% = I%
             CALL TurnOnMenu(I%)
             MenuCheckEvent$ = "menuopen"
          END IF
          NEXT I%  'For i% = 1 TO NumMenus%
      END IF        'Controllo lunghezza carattere
        EXIT FUNCTION
   END IF           'Controlla il tasto


   REM *** Controlla la pressione del pulsante sinistro

   Button% = 0
   InRegs.bx = 0
   InRegs.ax = 5
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   NumberTimes% = OutRegs.bx
   ScRow% = OutRegs.dx \ 8 + 1
   ScCol% = OutRegs.cx \ 8 + 1

   IF NumberTimes% <> 0 THEN
      IF MenusOnFlag% AND (ScRow% = 1) THEN
         MenubarCandidate% = ScCol% \ 10 + 1
         IF MenubarCandidate% <= NumMenus% THEN
            MenuNo% = MenubarCandidate%
            ChoiceNo% = 0
            MenuNowOpen% = MenubarCandidate%
            CALL TurnOnMenu(MenubarCandidate%)
            MenuCheckEvent$ = "menuopen"
            EXIT FUNCTION
         END IF
      ELSE
         MenuCheckEvent$ = "mousedown"
         EXIT FUNCTION
```

continua

Listato 4.5 *La funzione MenuCheckEvent$ per leggere le azioni senza attesa*

```
      END IF
   END IF


   REM *** Controlla il rilascio del pulsante sinistro


   InRegs.bx = 0
   InRegs.ax = 6
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   NumberTimes% = OutRegs.bx
   ScRow% = OutRegs.dx \ 8 + 1
   ScCol% = OutRegs.cx \ 8 + 1

   IF NumberTimes% <> 0 THEN
      IF MenusOnFlag% AND (MenuNowOpen% <> 0) THEN
         IF ScRow% >= 2 AND ScRow% <= 2 + Rows(MenuNowOpen%) THEN
            IF ScCol% >= 1 + 10 * (MenuNowOpen% - 1) AND ScCol%
                      <= 10 * (MenuNowOpen%) THEN
               ChoiceNo% = ScRow% - 1
               CALL TurnOffMenu(MenuNowOpen%)
               MenuNo% = MenuNowOpen%
               MenuNowOpen% = 0
               MenuCheckEvent$ = "menuchoice"
               EXIT FUNCTION
            ELSE
               MenuNo% = MenuNowOpen%
               CALL TurnOffMenu(MenuNowOpen%)
               MenuNowOpen% = 0
               MenuCheckEvent$ = "menuclose"
               EXIT FUNCTION
            END IF
         ELSE
            MenuNo% = MenuNowOpen%
            CALL TurnOffMenu(MenuNowOpen%)
            MenuNowOpen% = 0
            MenuCheckEvent$ = "menuclose"
            EXIT FUNCTION
         END IF
      ELSE
         MenuCheckEvent$ = "mouseup"
         EXIT FUNCTION
      END IF 'Menu open
   END IF 'NumberTimes% <> 0
```

continua

Listato 4.5 *La funzione MenuCheckEvent$ per leggere le azioni senza attesa*

```
    REM *** Controlla la pressione del pulsante destro

    Button% = 1
    InRegs.bx = 1
    InRegs.ax = 5
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    NumberTimes% = OutRegs.bx
    ScRow% = OutRegs.dx \ 8 + 1
    ScCol% = OutRegs.cx \ 8 + 1

    IF NumberTimes% <> 0 THEN
       MenuCheckEvent$ = "mousedown"
       EXIT FUNCTION
    END IF


    REM *** Controlla il rilascio del pulsante destro

    InRegs.bx = 1
    InRegs.ax = 6
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    NumberTimes% = OutRegs.bx
    ScRow% = OutRegs.dx \ 8 + 1
    ScCol% = OutRegs.cx \ 8 + 1

    IF NumberTimes% <> 0 THEN
       MenuCheckEvent$ = "mousedown"
       EXIT FUNCTION
    END IF

END FUNCTION

SUB TurnOffMenu (MNumber%)

    CurRow% = CSRLIN
    CurCol% = POS(0)
    DIM InRegs AS RegType, OutRegs AS RegType

    InRegs.ax = 2
    CALL INTERRUPT(&H33, InRegs, OutRegs)


    REM Stampa il testo precedente
```

continua

Listato 4.5 *La funzione MenuCheckEvent$ per leggere le azioni senza attesa*

```
    TR = TopRow(MNumber%)
    TC = TopCol(MNumber%)
    LOCATE TR, TC
    InRegs.cx = 1

    FOR I = 1 TO Rows(MNumber%)
       OR$ = OldText(MNumber%, I)
       OC$ = OldAttrb(MNumber%, I)
       FOR j = 1 TO Cols(MNumber%)
          InRegs.ax = &H900 + ASC(MID$(OR$, j, 1))
          InRegs.bx = ASC(MID$(OC$, j, 1))
          CALL INTERRUPT(&H10, InRegs, OutRegs)
          LOCATE TR + I - 1, TC + j
       NEXT j
          LOCATE TR + I, TC
    NEXT I

    InRegs.ax = 1
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    LOCATE CurRow%, CurCol%

END SUB

SUB TurnOnMenu (MNumber%)

    DIM InRegs AS RegType, OutRegs AS RegType

    CurRow% = CSRLIN
    CurCol% = POS(0)

    InRegs.ax = 2
    CALL INTERRUPT(&H33, InRegs, OutRegs)


    REM Salva la precedente sezione dello schermo

    rr = TopRow(MNumber%)
    cc = TopCol(MNumber%)
    rt = Rows(MNumber%)
    ct = Cols(MNumber%)

    FOR I = 1 TO rt
       temp1$ = ""
       temp2$ = ""
```

continua

Listato 4.5 *La funzione MenuCheckEvent$ per leggere le azioni senza attesa*

```
      FOR j = 1 TO ct
         temp1$ = temp1$ + CHR$(SCREEN(rr + I - 1, cc + j - 1))
         temp2$ = temp2$ + CHR$(SCREEN(rr + I - 1, cc + j
                                        - 1, 1))
      NEXT j
      OldText(MNumber%, I) = temp1$
      OldAttrb(MNumber%, I) = temp2$
   NEXT I

   REM Ora stampa il nuovo testo

   TR = TopRow(MNumber%)
   TC = TopCol(MNumber%)
   LOCATE TR, TC
   InRegs.cx = 1

   FOR I = 1 TO Rows(MNumber%)
      OR$ = Text(MNumber%, I)
      FOR j = 1 TO Cols(MNumber%)
         InRegs.ax = &H900 + ASC(MID$(OR$, j, 1))
         InRegs.bx = Attribute(MNumber%)
         CALL INTERRUPT(&H10, InRegs, OutRegs)
         LOCATE TR + I - 1, TC + j
      NEXT j
         LOCATE TR + I, TC
   NEXT I

   InRegs.ax = 1
   CALL INTERRUPT(&H33, InRegs, OutRegs)

   LOCATE CurRow%, CurCol%
END SUB
```

Listato 4.5 *La funzione MenuCheckEvent$ per leggere le azioni senza attesa.*

PROVA DI IMMISSIONI PROVENIENTI DAL MENU

Di seguito viene indicato come utilizzare *MenuCheckEvent$( )*.

```
DECLARE FUNCTION MenuInitialize% (MenuNames( ) AS STRING,
        MenuChoices( ) AS STRING, BarAttrb%, MenuAttrbs( )
        AS INTEGER)
DECLARE SUB MenuShow( )
DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE FUNCTION MenuGetEvent$(MenuNo%, ChoiceNo%, Button%,
        Row%, Col%)
```

```
DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
   MenuChoices(2, 1) = "Piselli"
   MenuChoices(2, 2) = "Grano"
   MenuChoices(2, 3) = "Broccoli"

   MenuNames(3) = "Carne"
   MenuChoices(3, 1) = "Pollo"
   MenuChoices(3, 2) = "Maiale"
   MenuChoices(3, 3) = "Manzo"
   MenuChoices(3, 4) = "Pesce"

   FOR i= 1 TO 3
      MenuAttrbs(i) = &H61
   NEXT i
   BarAttrb% = &H24
   Check% = MenuInitialize%(MenuNames( ), MenuChoices( ),⌉
            BarAttrb%, MenuAttrbs( ))
   IF Check% = 0 THEN
      PRINT "ERRORE!"
   ELSE
      PRINT "Menu impostato"
   END IF

   CALL MenuShow
   Check% = MouseInitialize%

   IF Check% = 0 THEN PRINT "Mouse non attivato"

   CALL MouseShowCursor

   PRINT "Selezionare una voce di menu."
   DO
   Check$ = MenuCheckEvent$(MenuNo%, ChoiceNo%, Button%, Row%,⌉
            Col%)
   LOOP UNTIL Check$ = "menuchoice"
   PRINT "La selezione effettuata è stata ",MenuChoices(MenuNo%,⌉
         ChoiceNo%)
```

In questo caso il programma di esempio è praticamente identico al programma di esempio precedentemente mostrato per *MenuGetEvent$( )* e sarà qui sufficiente attendere che a *MenuCheckEvent$( )* venga trasferita un'azione da menu perché questa venga regolarmente registrata e emessa a video.
I sistemi a menu, a volte, richiedono altre opzioni che li rendono più completi. Per esempio, vi potrebbero essere alcune opzioni con doppia funzione di "attivazione/disattivazione". Si supponga che un'opzione di un programma per l'elaborazione di testi preveda la funzione di controllo ortografico: se questa funzione è attivata allora verrà attivato anche il controllo ortografico sul testo digitato. Il prossimo paragrafo spiegherà come inserire in un menu questa possibilità.

MARCATURA DI UNA SELEZIONE DA MENU

Verrà ora sviluppato un programma (*MenuMarkChoice( )*) che sia in grado di contrassegnare (*marcare*) una selezione da menu facendo comparire a fianco della stessa un simbolo di contrassegno. La presenza del simbolo di contrassegno indica che una determinata opzione è attiva e la volta successiva che il menu verrà aperto, di fianco all'opzione verrà conservato il contrassegno di marcatura. Per esempio, si supponga che la funzione di controllo ortografico sia la quarta voce del sesto menu: l'invocazione di *MenuMarkChoice(6, 4)* assocerà tale voce al simbolo di contrassegno definito. In linea generale, *MenuMarkChoice( )* potrà essere definito come segue:

```
CALL MenuMarkChoice (MenuNumber%, ChoiceNumber%)
```

Di seguito vengono indicati i significati dei parametri della riga di richiamo:

Nome parametro	*Descrizione*
MenuNumber%	Numero del menu contenente la voce da contrassegnare.
ChoiceNumber%	Numero della voce da contrassegnare all'interno del menu indicato.

Questo sottoprogramma potrebbe essere semplicisticamente paragonato a *MenuGet Event$( )* oppure a *MenuCheckEvent$( )*. In effetti, tutto quello che il programma fa è solo di immettere un simbolo di contrassegno (corrispondente a *CHR$(251)*) come primo carattere della riga contenente la voce da contrassegnare del menu che la contiene. Poiché le righe dei menu vengono memorizzate sotto forma di stringa in *Text(i, j)*, dove *i* corrisponde al numero del menu e *j* al numero della voce da contrassegnare, l'impostazione dovrebbe essere simile a quella di seguito riportata.

```
MID$(Text(MenuNumber%, ChoiceNumber%),1,1) = CHR$(251)
```

> ***Consiglio:*** Poiché questo codice è molto breve, esso potrà essere direttamente immesso all'interno di un programma, invece di associarvelo mediante operazione di *linking*.

Il listato 4.6 mostra l'intero programma. Notare che sarà ancora necessario includere le istruzioni *COMMON* per i menu.

```
DECLARE SUB MenuUnMarkChoice (MenuNumber%, ChoiceNumber%)

    CONST MaxMenus = 7
    DIM Rows(1 TO MaxMenus) AS INTEGER
    DIM Cols(1 TO MaxMenus) AS INTEGER
    DIM TopRow(1 TO MaxMenus) AS INTEGER
    DIM TopCol(1 TO MaxMenus) AS INTEGER
    DIM BotRow(1 TO MaxMenus) AS INTEGER
    DIM BotCol(1 TO MaxMenus) AS INTEGER
    DIM Attribute(1 TO MaxMenus) AS INTEGER
    DIM Text(1 TO MaxMenus, 25) AS STRING
    DIM OldText(1 TO MaxMenus, 25) AS STRING
    DIM OldAttrb(1 TO MaxMenus, 25) AS STRING
    COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,
         TopRow() AS INTEGER, TopCol() AS INTEGER, BotRow()
         AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /MenuB/ Attribute() AS INTEGER, Text()
         AS STRING, OldText() AS STRING, OldAttrb() AS STRING
    COMMON SHARED /MenuC/ Bar$, BarAttrb, MenusOnFlag%,
         NumMenus%, OldBar$, OldBarAttrb$

SUB MenuUnMarkChoice (MenuNumber%, ChoiceNumber%)

    MID$(Text(MenuNumber%, ChoiceNumber%),1,1) = " "

END SUB
```

Listato 4.6 *Il sottoprogramma MenuMarkChoice per contrassegnare le voci di un menu*

Da notare che la voce di menu non viene visualizzata immediatamente dopo la sua selezione, vale a dire che *MenuMarkChoice()* non visualizzerà il menu con il contrassegno a fianco della voce selezionata e questo poiché l'attivazione di una voce è un'azione da menu: per esempio, quando viene selezionata la funzione di controllo ortografico, il menu verrà chiuso. Per controllare la corretta selezione di una voce sarà quindi necessario riaprire il menu che la contiene.

PROVA DI MARCATURA DI VOCI NEL MENU

In questo esempio si potrà attivare una qualsiasi delle voci di un menu senza, tuttavia, avere ancora la possibilità di disattivarle.

```
DECLARE SUB MenuMarkChoice (MenuNumber%, ChoiceNumber%)

DECLARE FUNCTION MenuInitialize% (MenuNames( ) AS STRING,
        MenuChoices( ) AS STRING, BarAttrb%, MenuAttrbs( )
        AS INTEGER)
DECLARE SUB MenuShow( )
DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE FUNCTION MenuGetEvent$(MenuNo%, ChoiceNo%, Button%,
        Row%, Col%)

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
   MenuChoices(2, 1) = "Piselli"
   MenuChoices(2, 2) = "Grano"
   MenuChoices(2, 3) = "Broccoli"

   MenuNames(3) = "Carne"
   MenuChoices(3, 1) = "Pollo"
   MenuChoices(3, 2) = "Maiale"
   MenuChoices(3, 3) = "Manzo"
   MenuChoices(3, 4) = "Pesce"

   FOR i= 1 TO 3
      MenuAttrbs(i) = &H61
   NEXT i
   BarAttrb% = &H24

   Check% = MenuInitialize%(MenuNames( ), MenuChoices( ),
            BarAttrb%, MenuAttrbs( ))
   IF Check% = 0 THEN
      PRINT "ERRORE!"
   ELSE
```

```
    PRINT "Menu impostato"
END IF

CALL MenuShow

Check% = MouseInitialize%

IF Check% = 0 THEN PRINT "Mouse non attivato"

CALL MouseShowCursor

PRINT "Selezionare una voce da contrassegnare."
PRINT "Premere E per uscire."
DO
Check$ = MenuGetEvent$(MenuNo%, ChoiceNo%, Button%,⌉
         Row%, Col%)
IF Check$ = "menuchoice" THEN CALL MenuMarkChoice(MenuNo%,⌉
             ChoiceNo%)
LOOP UNTIL UCASE$(Check$) = "E"
```

Tutto ciò che viene richiesto di fare all'utente è di selezionare una voce di menu:

```
PRINT "Selezionare una voce da contrassegnare."
PRINT "Premere E per uscire."
:
```

A questo punto verrà attivato un ciclo fino a quando non verrà effettuata una selezione e la relativa voce non verrà contrassegnata da *MenuMarkChoice( )* (almeno fino a quando non verrà premuto *E* o per uscire).

```
PRINT "Selezionare una voce da contrassegnare."
PRINT "Premere E per uscire."
   DO
   Check$ = MenuGetEvent$(MenuNo%, ChoiceNo%, Button%, Row%,⌉
                          Col%)
   IF Check$ = "menuchoice" THEN CALL MenuMarkChoice(MenuNo%,⌉
             ChoiceNo%)
   LOOP UNTIL UCASE$(Check$) = "E"
```

Vista ora la possibilità di contrassegnare una voce di menu, il logico passo successivo sarà quello di creare una procedura per poterla disattivare.

DISATTIVAZIONE DELLA MARCATURA DI UNA VOCE DI MENU

Questo programma elimina il contrassegno (*marcatura*) a una voce di menu precedentemente contrassegnata, vale a dire che se una voce di menu è stata associata al simbolo di contrassegno, *MenuUnMarkChoice( )* lo eliminerà disattivando, nel contempo, la marcatura della voce e indicando quindi che la voce non è più attiva. Di seguito viene indicato come avviare la procedura:

CALL MenuUnMarkChoice (MenuNumber%, ChoiceNumber%)

dove:

Nome parametro	*Descrizione*
MenuNumber%	corrisponde al numero del menu con la voce da disattivare,
ChoiceNumber%	corrisponde alla voce da disattivare contenuta all'interno del menu precedentemente indicato.

Come già detto per *MenuMarkChoice( )* il listato è anche in questo caso molto semplice e può essere contenuto in una sola riga:

```
MID$(Text(MenuNumber%, ChoiceNumber%),1,1) = " "
```

Questa riga di istruzione non fa altro che eliminare il contrassegno posto a fianco della voce precedentemente selezionata. Il listato 4.7 presenta il sottoprogramma.

```
DECLARE SUB MenuUnMarkChoice (MenuNumber%, ChoiceNumber%)
    CONST MaxMenus = 7
    DIM Rows(1 TO MaxMenus) AS INTEGER
    DIM Cols(1 TO MaxMenus) AS INTEGER
    DIM TopRow(1 TO MaxMenus) AS INTEGER
    DIM TopCol(1 TO MaxMenus) AS INTEGER
    DIM BotRow(1 TO MaxMenus) AS INTEGER
    DIM BotCol(1 TO MaxMenus) AS INTEGER
    DIM Attribute(1 TO MaxMenus) AS INTEGER
    DIM Text(1 TO MaxMenus, 25) AS STRING
    DIM OldText(1 TO MaxMenus, 25) AS STRING
    DIM OldAttrb(1 TO MaxMenus, 25) AS STRING
    COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,
         TopRow() AS INTEGER, TopCol() AS INTEGER, BotRow()
         AS INTEGER, BotCol() AS INTEGER
```

continua

Listato 4.7 *Il sottoprogramma MenuUnMarkChoice per eliminare il contrassegno a una voce di menu*

```
    COMMON SHARED /MenuB/ Attribute() AS INTEGER, Text()
         AS STRING, OldText() AS STRING, OldAttrb() AS STRING
    COMMON SHARED /MenuC/ Bar$, BarAttrb, MenusOnFlag%,
         NumMenus%, OldBar$, OldBarAttrb$

SUB MenuUnMarkChoice (MenuNumber%, ChoiceNumber%)

    MID$(Text(MenuNumber%, ChoiceNumber%),1,1) = " "

END SUB
```

Listato 4.7 *Il sottoprogramma MenuUnMarkChoice per eliminare il contrassegno a una voce di menu*

PROVA DI DISATTIVAZIONE DI VOCI DEL MENU

Di seguito viene riportato come utilizzare i due sottoprogrammi *MenuMarkChoice( )* e *MenuUnMarkChoice( )*.

```
DECLARE FUNCTION MenuInitialize% (MenuNames( ) AS STRING,
        MenuChoices( ) AS STRING, BarAttrb%, MenuAttrbs( )
        AS INTEGER)
DECLARE SUB MenuShow( )
DECLARE FUNCTION MouseInitialize% ( )
DECLARE SUB MouseShowCursor ( )
DECLARE FUNCTION MenuGetEvent$(MenuNo%, ChoiceNo%, Button%,
                               Row%, Col%)
DECLARE SUB MenuMarkChoice (MenuNumber%, ChoiceNumber%)
DECLARE SUB MenuUnMarkChoice (MenuNumber%, ChoiceNumber%)

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
   MenuChoices(2, 1) = "Piselli"
   MenuChoices(2, 2) = "Grano"
   MenuChoices(2, 3) = "Broccoli"
```

```
MenuNames(3) = "Carne"
MenuChoices(3, 1) = "Pollo"
MenuChoices(3, 2) = "Maiale"
MenuChoices(3, 3) = "Manzo"
MenuChoices(3, 4) = "Pesce"

FOR i= 1 TO 3
   MenuAttrbs(i) = &H61
NEXT i
BarAttrb% = &H24

Check% = MenuInitialize%(MenuNames( ), MenuChoices( ),
     BarAttrb%, MenuAttrbs( ))
IF Check% = 0 THEN
   PRINT "ERRORE!"
ELSE
   PRINT "Menu impostato"
END IF

CALL MenuShow

Check% = MouseInitialize%

IF Check% = 0 THEN PRINT "Mouse non attivato"

CALL MouseShowCursor

PRINT "Selezionare una voce da contrassegnare."
PRINT "Premere E per uscire."

DO
Check$ = MenuGetEvent$(MenuNo%, ChoiceNo%, Button%, Row%, Col%)
IF Check$ = "menuchoice" THEN CALL MenuMarkChoice(MenuNo%,
     ChoiceNo%)
LOOP UNTIL UCASES(Check$) = "E"

PRINT "Ora selezionare una voce da disattivare."
PRINT "Premere E per uscire."

DO
   Check$ = MenuGetEvent$(MenuNo%, ChoiceNo%, Button%, Row%, Col%)
   IF Check$ = "menuchoice" THEN CALL MenuUnMarkChoice(MenuNo%,
        ChoiceNo%)
   LOOP UNTIL UCASES(Check$) = "E"
```

Questo programma di esempio richiede prima di marcare una o più voci, proprio come si è fatto con il precedente programma:

```
PRINT "Selezionare una voce da contrassegnare."
PRINT "Premere E per uscire."

DO
Check$ = MenuGetEvent$(MenuNo%, ChoiceNo%, Button%, Row%, Col%)
IF Check$ = "menuchoice" THEN CALL MenuMarkChoice(MenuNo%,⌉
     ChoiceNo%)
LOOP UNTIL UCASES(Check$) = "E"
:
:
```

A questo punto sarà necessario premere *E* per uscire. Vi è ora la possibilità di deselezionare la/le voci precedentemente contrassegnate.

```
PRINT "Selezionare una voce da contrassegnare."
PRINT "Premere E per uscire."

DO
Check$ = MenuGetEvent$(MenuNo%, ChoiceNo%, Button%, Row%, Col%)
IF Check$ = "menuchoice" THEN CALL MenuMarkChoice(MenuNo%,⌉
     ChoiceNo%)
LOOP UNTIL UCASES(Check$) = "E"

PRINT "Ora selezionare una voce da disattivare."
PRINT "Premere E per uscire."

DO
Check$ = MenuGetEvent$(MenuNo%, ChoiceNo%, Button%, Row%, Col%)
IF Check$ = "menuchoice" THEN CALL MenuUnMarkChoice(MenuNo%,⌉
     ChoiceNo%)
LOOP UNTIL UCASES(Check$) = "E"
```

Anche in questo caso si potrà premere *E* per uscire.
Vi è ancora un'ulteriore routine che richiede di essere sviluppata: se si sta utilizzando *MenuGetEvent$( )* o *MenuCheckEvent( )* sarà necessario collegare le immissioni da menu a *MenuNo%* e *ChoiceNo%*.

```
InString$ = MenuGetEvent$ (MenuNo%, ChoiceNo%, Button%, ScRow%,⌉
                           ScCol%)
```

Vi è tuttavia la possibilità che il contenuto degli array originali sia andato perso. In questo caso, si dovrebbe essere in grado di interrogare il sistema a menu per controllare, per esempio, a cosa corrisponde *Menu 5* e *Choice 3*. Viene perciò qui presentata l'ultima funzione del sistema a menu a cui verrà assegnato il nome *MenuReadChoice$( )*.

CONTROLLO DI UN MENU

Se si intende controllare a cosa corrisponde una selezione di una voce di un menu, per esempio la selezione della voce corrispondente a *MenuNo%* e *ChoiceNo%* restituita da *MenuGetEvent%( )*, sarà necessario richiamare una funzione a cui si potrà assegnare il nome *MenuReadChoice$(MenuNo%, ChoiceNo%)*.
A questa funzione, inoltre, si potrà aggiungere un'ulteriore caratteristica che prevede, se viene trasferito un numero di menu pari a *0*, l'emissione da parte di *MenuReadChoice$( )* del nome del menu prelevato dalla barra dei menu. Per esempio, *MenuReadChoice$(0, 2)* dovrebbe restituire il nome del secondo menu presente sulla barra dei menu (figura 4.8).

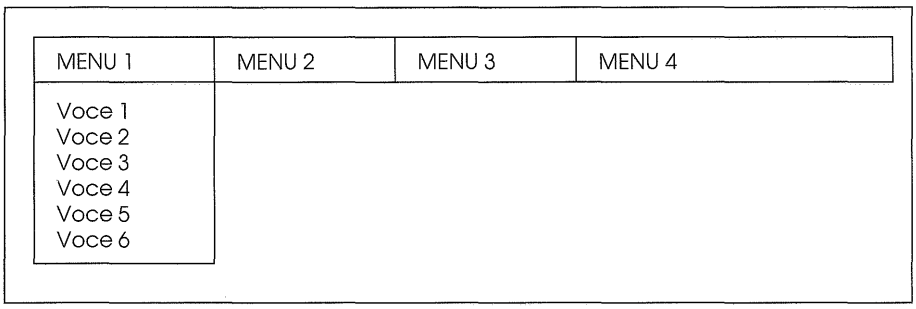

Figura 4.8

Di seguito viene indicato come utilizzare la funzione *MenuReadChoice$( )*.

`ReadString$ = MenuReadChoice$ (menuNo%, ChoiceNo%)`

Nome parametro	*Descrizione*
MenuNo%	corrisponde al numero del menu (trasferito a *MenuInitialize%( )*) che contiene la voce da controllare,
ChoiceNo%	corrisponde alla voce contenuta all'interno del menu da controllare.

Quanto sopra dovrebbe restituire:

Funzione	*Valore restituito*	*Descrizione*
MenuReadChoice$	MenuNo% = 0	*MenuReadChoice$* = nome del menu prelevato dalla barra dei menu (il cui numero è stato trasferito a *ChoiceNo%*).

MenuNo% = 1	*MenuReadChoice$* = nome della voce *n* del menu *m* (dove *n* corrisponde al numero della voce trasferita a *ChoiceNo%* e *m* corrisponde al numero trasferito a *ChoiceNo%*).

La realizzazione del programma non è molto complessa: viene in sostanza richiesto che il programma restituisca un nome di menu e della relativa voce. La restituzione del nome del menu viene definita da:

```
IF MenuNo% = 0 THEN
   MenuReadChoice$ = RTRIM$(MID$(Bar$, (ChoiceNo%-1)*10
                                + 2, 8))
   :
   :
```

Verrà ora utilizzata la funzione *RTRIM$( )* del BASIC che elimina eventuali spazi di riempimento associati al nome del menu. Se non viene attivata questa funzione sarà necessario immettere la voce estratta in un particolare menu. L'associazione di *LTRIM$( )* e *RTRIM$( )* può essere utilizzata come di seguito riportato:

```
IF MenuNo% = 0 THEN
   MenuReadChoice$ = RTRIM$(MID$(Bar$, (ChoiceNo%-1)*10
                                + 2, 8))
ELSE
   MenuReadChoice$ = RTRIM$(LTRIM$(Text(MenuNo%, ChoiceNo%)))
END IF
:
:
```

Il listato 4.8 presenta la funzione completa.

```
DECLARE FUNCTION MenuReadChoice$ (MenuNo%, ChoiceNo%)

    CONST MaxMenus = 7
    DIM Rows(1 TO MaxMenus) AS INTEGER
    DIM Cols(1 TO MaxMenus) AS INTEGER
    DIM TopRow(1 TO MaxMenus) AS INTEGER
    DIM TopCol(1 TO MaxMenus) AS INTEGER
    DIM BotRow(1 TO MaxMenus) AS INTEGER
    DIM BotCol(1 TO MaxMenus) AS INTEGER
    DIM Attribute(1 TO MaxMenus) AS INTEGER
    DIM Text(1 TO MaxMenus, 25) AS STRING
    DIM OldText(1 TO MaxMenus, 25) AS STRING
    DIM OldAttrb(1 TO MaxMenus, 25) AS STRING
```

continua

Listato 4.8 *La funzione MenuReadChoice$( )*

```
    COMMON SHARED /MenuA/ Rows() AS INTEGER, Cols() AS INTEGER,
         TopRow() AS INTEGER, TopCol() AS INTEGER, BotRow()
         AS INTEGER, BotCol() AS INTEGER
    COMMON SHARED /MenuB/ Attribute() AS INTEGER, Text()
         AS STRING, OldText() AS STRING, OldAttrb() AS STRING
    COMMON SHARED /MenuC/ Bar$, BarAttrb, MenusOnFlag%,
         NumMenus%, OldBar$, OldBarAttrb$

FUNCTION MenuReadChoice$ (MenuNo%, ChoiceNo%)

    IF MenuNo% = 0 THEN
       MenuReadChoice$ = RTRIM$(MID$(Bar$, (ChoiceNo%-1)*10
                                 + 2, 8))
    ELSE
       MenuReadChoice$ = RTRIM$(LTRIM$(Text(MenuNo%, ChoiceNo%)))
    END IF

END FUNCTION
```

Listato 4.8 *La funzione MenuReadChoice$( ).*

PROVA DI LETTURA DELLE VOCI DEL MENU

Viene ora riportato il programma che consente la lettura delle voci del menu di prova precedentemente creato.

```
DECLARE FUNCTION MenuInitialize% (MenuNames( ) AS STRING,
        MenuChoices( ) AS STRING, BarAttrb%, MenuAttrbs( )
        AS INTEGER)
DECLARE FUNCTION MenuReadChoice$ (MenuNo%, ChoiceNo%)

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
   MenuChoices(2, 1) = "Piselli"
   MenuChoices(2, 2) = "Grano"
   MenuChoices(2, 3) = "Broccoli"
```

```
MenuNames(3) = "Carne"
MenuChoices(3, 1) = "Pollo"
MenuChoices(3, 2) = "Maiale"
MenuChoices(3, 3) = "Manzo"
MenuChoices(3, 4) = "Pesce"

FOR i= 1 TO 3
   MenuAttrbs(i) = &H61
NEXT i
BarAttrb% = &H24

Check% = MenuInitialize%(MenuNames( ), MenuChoices( ),
      BarAttrb%, MenuAttrbs( ))
IF Check% = 0 THEN
   PRINT "ERRORE!"
ELSE
   PRINT "Menu impostato"
END IF

PRINT "La voce 1 del menu 1 è: ", MenuReadChoice(1, 1)
PRINT "Il menu numero 1 è: ", MenuReadChoice(0, 1)
```

Il programma sopra riportato imposta un sistema a menu e indica il contenuto della voce 1 del menu 1 oltre a indicare il nome del menu stesso.
Si è visto così come impostare un sistema a menu; come visualizzare o nascondere la barra dei menu; come leggere le immissioni provenienti da tastiera, mouse o menu; come attivare/disattivare una particolare voce di un menu e come interrogare il sistema a menu in modo che venga restituita una voce particolare di menu.
Se si possiede il BASIC PDS il prossimo paragrafo presenterà alcuni sistemi a menu precostituiti.

IL SISTEMA A MENU DEL BASIC PDS

Il BASIC PDS contiene una serie di funzioni precostituite che sono in un qualche modo simili a quelle che sono state realizzate nei paragrafi precedenti. Si consiglia, per cui, di provare queste funzioni. Di seguito verrà creato un programma di esempio del tipo di quello che è stato precedentemente sviluppato per *MenuGetEvent$( )* che visualizza la barra dei menu e che riporta quale selezione è stata effettuata all'interno di un particolare menu.

***Nota**: Anche se non si possiede il BASIC PDS la lettura di questo paragrafo potrebbe essere comunque utile per conoscere cosa viene da esso offerto.*

Si noti che se si utilizza QBX.EXE sarà necessario caricare anche la libreria che è stata creata nel capitolo 2 (*UIASM.QLB*). Se si volesse, per esempio, richiamare il programma dimostrativo *MENUTEST.BAS*, QBX.EXE dovrebbe essere avviato come di seguito riportato:

```
QBX MENUTEST \L UIASM
```

Inoltre, è necessario caricare anche MENU.BAS, WINDOW.BAS, MOUSE.BAS e GENERAL.BAS poiché il sistema a menu del BASIC PDS utilizzerà alcuni codici contenuti in tutti i file sopra riportati.

> ***Consiglio***: QuickBASIC (*QB.EXE* oppure *QBX.EXE*) consente di caricare contemporaneamente più file utilizzando i file con estensione *.MAK*. Per produrre un file *.MAK* è necessario caricare i moduli appropriati e selezionare l'opzione di salvataggio dal menu per la gestione dei file. La volta successiva che verrà caricato il modulo principale verranno automaticamente caricati anche tutti i file ad esso collegati.

D'altra parte, se viene utilizzato *BC.EXE* sarà necessario, invece, includere nell'elenco delle librerie da collegare anche la libreria *UIASM.LIB* precedentemente creata.

```
' $INCLUDE: 'C:\BC7\GENERAL.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\WINDOW.BI'
   :
   :
```

Il passo successivo sarà quello di impostare le istruzioni COMMON e dimensionare gli array:

```
' $INCLUDE: 'C:\BC7\GENERAL.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\WINDOW.BI'

COMMON SHARED /uitools/ GloMenu          AS MenuMiscType
COMMON SHARED /uitools/ GloTitle( )      AS MenuTitleType
COMMON SHARED /uitools/ GloItem( )       AS MenuItemType
DIM GloTitle(MAXMENU)                    AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)            AS MenuItemType
   :
   :
```

Per rendere il programma simile a uno di quelli che sono già stati presentati per la funzione *menuGetEvent$( )*, sarà ora necessario dimensionare gli array dei menu e delle voci all'interno degli stessi come di seguito riportato:

```
' $INCLUDE: 'C:\BC7\GENERAL.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\WINDOW.BI'

COMMON SHARED /uitools/ GloMenu            AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()         AS MenuTitleType
COMMON SHARED /uitools/ GloItem()          AS MenuItemType

DIM GloTitle(MAXMENU)            AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)    AS MenuItemType

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER
   :
   :
```

e riempirli come si è fatto in precedenza:

```
' $INCLUDE: 'C:\BC7\GENERAL.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\WINDOW.BI'

COMMON SHARED /uitools/ GloMenu            AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()         AS MenuTitleType
COMMON SHARED /uitools/ GloItem()          AS MenuItemType

DIM GloTitle(MAXMENU)            AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)    AS MenuItemType

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
   MenuChoices(2, 1) = "Piselli"
   MenuChoices(2, 2) = "Grano"
   MenuChoices(2, 3) = "Broccoli"

   MenuNames(3) = "Carne"
```

```
MenuChoices(3, 1) = "Pollo"
MenuChoices(3, 2) = "Maiale"
MenuChoices(3, 3) = "Manzo"
MenuChoices(3, 4) = "Pesce"

FOR i = 1 TO 3
   MenuAttrbs(i) = &H61
NEXT i
BarAttrb% = &H24
:
:
```

Ora si è pronti per utilizzare il sistema a menu del BASIC PDS. Innanzitutto si inizializzerà il sistema con un'invocazione al sottoprogramma PDS *MenuInit( )* e in seguito i nomi e le opzioni di menu potranno essere caricati con *MenuSet( )*. *MenuSet( )* potrà essere utilizzato come di seguito riportato:

CALL MenuSet(Menu%, Item%, State%, Text$, Accesskey%)

Nome parametro	*Descrizione*
Menu%	corrisponde al numero del menu.
Item%	corrisponde al numero della voce all'interno del menu (0 = nome del menu).
State%	0 = la selezione è "disabilitata" (per esempio, selezione non valida); 1 = la selezione è "abilitata" (stato normale); 2 = la selezione è "abilitata" e attivata con un segno di contrassegno.
Text$	rappresenta il nome del menu o della selezione stessa.
Accesskey%	corrisponde alla posizione del carattere in *Text$* che viene considerato come lettera per avviare la voce. Questa lettera è normalmente evidenziata all'interno del menu.

Questo significa che sarà necessario avviare un *loop* su *menuSet( )* richiamandolo per ciascuna selezione di menu. Di seguito viene riportato come caricare i dati dagli array e trasferirli all'interno del sistema a menu del BASIC PDS.

```
' $INCLUDE: 'C:\BC7\GENERAL.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\WINDOW.BI'

COMMON SHARED /uitools/ GloMenu          AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()       AS MenuTitleType
COMMON SHARED /uitools/ GloItem()        AS MenuItemType

DIM GloTitle(MAXMENU)            AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)    AS MenuItemType
```

```
DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
   MenuChoices(2, 1) = "Piselli"
   MenuChoices(2, 2) = "Grano"
   MenuChoices(2, 3) = "Broccoli"

   MenuNames(3) = "Carne"
   MenuChoices(3, 1) = "Pollo"
   MenuChoices(3, 2) = "Maiale"
   MenuChoices(3, 3) = "Manzo"
   MenuChoices(3, 4) = "Pesce"

   FOR i = 1 TO 3
      MenuAttrbs(i) = &H61
   NEXT i
   BarAttrb% = &H24

   CALL MenuInit

   FOR i% = 1 TO 3
      CALL MenuSet(i%, 0, 1, MenuNames(i%), 1)
      FOR j% = 1 TO 5
         IF MenuChoices(i%, j%) <> "" THEN
            CALL MenuSet(i%, j%, 1, MenuChoices(i%, j%), 1)
         END IF
      NEXT j%
   NEXT i%
   :
   :
```

A questo punto si potrà richiamare il sottoprogramma PDS *MenuColor( )* per impostare lo schema dei colori:

CALL MenuColor(Fore%, Back%, High%, Disab%, Curfore%, Curback%, CurHi%)

Nome parametro	*Descrizione*
Fore%	colore di primo piano del menu (0-15).

Back%	colore dello sfondo del menu (0-7).
High%	colore evidenziato della lettera di avvio della selezione (0-15).
Disab%	colore del testo delle selezioni non attivabili (0-15).
Curfore%	colore di primo piano del puntatore del menu (0-7).
Curback%	colore dello sfondo del puntatore del menu (0-7).
CurHi%	colore del cursore del menu quando questo è posizionato sulla lettera evidenziata (0-15).

Dopo *MenuColor( )*, il passo successivo sarà quello di richiamare *MenuPreProcess( )* per poter costruire gli indici interni necessari al PDS per avviare i vari menu.

```
' $INCLUDE: 'C:\BC7\GENERAL.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\WINDOW.BI'

COMMON SHARED /uitools/ GloMenu            AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()         AS MenuTitleType
COMMON SHARED /uitools/ GloItem()          AS MenuItemType

DIM GloTitle(MAXMENU)            AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)    AS MenuItemType
DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
   MenuChoices(2, 1) = "Piselli"
   MenuChoices(2, 2) = "Grano"
   MenuChoices(2, 3) = "Broccoli"

   MenuNames(3) = "Carne"
   MenuChoices(3, 1) = "Pollo"
   MenuChoices(3, 2) = "Maiale"
   MenuChoices(3, 3) = "Manzo"
   MenuChoices(3, 4) = "Pesce"

   FOR i = 1 TO 3
```

```
      MenuAttrbs(i) = &H61
   NEXT i
   BarAttrb% = &H24

   CALL MenuInit

   FOR i% = 1 TO 3
      CALL MenuSet(i%, 0, 1, MenuNames(i%), 1)
      FOR j% = 1 TO 5
         IF MenuChoices(i%, j%) <> "" THEN
            CALL MenuSet(i%, j%, 1, MenuChoices(i%, j%), 1)
         END IF
      NEXT j%
   NEXT i%

   CALL MenuColor(14, 1, 12, 8, 14, 3, 12)

   CALL MenuPreProcess
   :
   :
```

È ora arrivato il momento di visualizzare la barra dei menu. Verrà qui utilizzato *MenuShow( )* e *MouseShow( )* in modi da visualizzare anche il puntatore del mouse e per poter emettere in seguito il messaggio "Effettuare una selezione dal menu".

```
' $INCLUDE: 'C:\BC7\GENERAL.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\WINDOW.BI'

COMMON SHARED /uitools/ GloMenu            AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()         AS MenuTitleType
COMMON SHARED /uitools/ GloItem()          AS MenuItemType

DIM GloTitle(MAXMENU)            AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)    AS MenuItemType

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
```

```
MenuChoices(2, 1) = "Piselli"
MenuChoices(2, 2) = "Grano"
MenuChoices(2, 3) = "Broccoli"

MenuNames(3) = "Carne"
MenuChoices(3, 1) = "Pollo"
MenuChoices(3, 2) = "Maiale"
MenuChoices(3, 3) = "Manzo"
MenuChoices(3, 4) = "Pesce"

FOR i = 1 TO 3
   MenuAttrbs(i) = &H61
NEXT i
BarAttrb% = &H24

CALL MenuInit

FOR i% = 1 TO 3
   CALL MenuSet(i%, 0, 1, MenuNames(i%), 1)
   FOR j% = 1 TO 5
      IF MenuChoices(i%, j%) <> "" THEN
         CALL MenuSet(i%, j%, 1, MenuChoices(i%, j%), 1)
      END IF
   NEXT j%
NEXT i%

CALL MenuColor(14, 1, 12, 8, 14, 3, 12)

CALL MenuPreProcess

CALL MenuShow

CALL MouseShow

LOCATE 2, 1
PRINT "Effettuare una selezione dal menu."
:
:
```

A questo punto a video dovrebbe essere visibile la barra dei menu e si sarà pronti a ricevere le immissioni. Il sistema a menu del BASIC PDS utilizza un metodo simile a quello già usato dal sistema a menu precedentemente creato: vi è qui una funzione, *MenuInkey$( )* che è molto simile a *MenuCheckEvent$( )*. Per poter trasferire le immissioni da tastiera o da menu (ma non da mouse) è necessario generare un *loop* su *MenuInkey$( )* fino a quando non verrà restituita una stringa di menu. In seguito si potrà interrogare il sistema a menu per controllare quale selezione è quella correntemente attiva:

```
' $INCLUDE: 'C:\BC7\GENERAL.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\WINDOW.BI'

COMMON SHARED /uitools/ GloMenu          AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()       AS MenuTitleType
COMMON SHARED /uitools/ GloItem()        AS MenuItemType

DIM GloTitle(MAXMENU)            AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)    AS MenuItemType

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

   MenuNames(1) = "Frutta"
   MenuChoices(1, 1) = "Mele"
   MenuChoices(1, 2) = "Banane"
   MenuChoices(1, 3) = "Uva"
   MenuChoices(1, 4) = "Pesche"
   MenuChoices(1, 5) = "Arance"

   MenuNames(2) = "Verdura"
   MenuChoices(2, 1) = "Piselli"
   MenuChoices(2, 2) = "Grano"
   MenuChoices(2, 3) = "Broccoli"

   MenuNames(3) = "Carne"
   MenuChoices(3, 1) = "Pollo"
   MenuChoices(3, 2) = "Maiale"
   MenuChoices(3, 3) = "Manzo"
   MenuChoices(3, 4) = "Pesce"

   FOR i = 1 TO 3
      MenuAttrbs(i) = &H61
   NEXT i
   BarAttrb% = &H24

   CALL MenuInit

   FOR i% = 1 TO 3
      CALL MenuSet(i%, 0, 1, MenuNames(i%), 1)
      FOR j% = 1 TO 5
         IF MenuChoices(i%, j%) <> "" THEN
            CALL MenuSet(i%, j%, 1, MenuChoices(i%, j%), 1)
         END IF
      NEXT j%
```

```
NEXT i%

CALL MenuColor(14, 1, 12, 8, 14, 3, 12)

CALL MenuPreProcess

CALL MenuShow

CALL MouseShow

LOCATE 2, 1
PRINT "Effettuare una selezione dal menu."

DO
Instring$ = MenuInkey$
LOOP WHILE Instring$ <> "menu"
:
:
```

Al termine di questo ciclo sarà stata effettuata una selezione di una voce di menu; la funzione *MenuCheck(0)* del BASIC PDS restituirà il numero del menu entro il quale è stata effettuata una selezione e *MenuCheck(1)* restituirà il numero della voce selezionata all'interno dello stesso menu. Questo significa che sarà possibile riportare a video il risultato delle selezione effettuata. Il listato 4.9 riporta il codice completo.

```
' $INCLUDE: 'C:\BC7\GENERAL.BI'
' $INCLUDE: 'C:\BC7\MOUSE.BI'
' $INCLUDE: 'C:\BC7\MENU.BI'
' $INCLUDE: 'C:\BC7\WINDOW.BI'

COMMON SHARED /uitools/ GloMenu          AS MenuMiscType
COMMON SHARED /uitools/ GloTitle()       AS MenuTitleType
COMMON SHARED /uitools/ GloItem()        AS MenuItemType

DIM GloTitle(MAXMENU)            AS MenuTitleType
DIM GloItem(MAXMENU, MAXITEM)    AS MenuItemType

DIM MenuNames(1 TO 3) AS STRING
DIM MenuChoices(1 TO 3, 1 TO 5) AS STRING
DIM MenuAttrbs(1 TO 3) AS INTEGER

    MenuNames(1) = "Frutta"
    MenuChoices(1, 1) = "Mele"
    MenuChoices(1, 2) = "Banane"
    MenuChoices(1, 3) = "Uva"
```

continua

Listato 4.9 *Il sistema a menu del BASIC PDS*

```
MenuChoices(1, 4) = "Pesche"
MenuChoices(1, 5) = "Arance"

MenuNames(2) = "Verdura"
MenuChoices(2, 1) = "Piselli"
MenuChoices(2, 2) = "Grano"
MenuChoices(2, 3) = "Broccoli"

MenuNames(3) = "Carne"
MenuChoices(3, 1) = "Pollo"
MenuChoices(3, 2) = "Maiale"
MenuChoices(3, 3) = "Manzo"
MenuChoices(3, 4) = "Pesce"

FOR i = 1 TO 3
   MenuAttrbs(i) = &H61
NEXT i
BarAttrb% = &H24
CALL MenuInit
FOR i% = 1 TO 3
   CALL MenuSet(i%, 0, 1, MenuNames(i%), 1)
   FOR j% = 1 TO 5
      IF MenuChoices(i%, j%) <> "" THEN
         CALL MenuSet(i%, j%, 1, MenuChoices(i%, j%), 1)
      END IF
   NEXT j%
NEXT i%

CALL MenuColor(14, 1, 12, 8, 14, 3, 12)
CALL MenuPreProcess
CALL MenuShow
CALL MouseShow

LOCATE 2, 1
PRINT "Effettuare una selezione dal menu."

DO
Instring$ = MenuInkey$
LOOP WHILE Instring$ <> "menu"
MenuNo% = MenuCheck(0)
ChoiceNo% = MenuCheck(1)
LOCATE 3, 1
PRINT "La selezione è: ", MenuChoices(MenuNo%, ChoiceNo%)
```

Listato 4.9 *Il sistema a menu del BASIC PDS.*

A questo punto, dopo aver effettuato una selezione da un menu, questa è stata decifrata dal programma di esempio e ne è stato visualizzato il risultato. Fondamen-

talmente il sistema a menu del BASIC PDS è molto simile a quello che è stato sviluppato nei paragrafi precedenti di questo capitolo con la sola restrizione che questo dovrà essere utilizzato per leggere tutte le immissioni all'interno del programma ma, in linea generale, questa procedura funziona abbastanza bene.

CONCLUSIONE

In questo capitolo è stato generato un sistema a menu e si sono analizzate le procedure attivabili con il BASIC PDS. Con questi sistemi sarà quindi possibile generare una serie di menu a tendina da applicare ai propri programmi utilizzando poche e semplici invocazioni.
Il capitolo 5 prenderà in considerazione la gestione della grafica e il suo utilizzo con QuickBASIC.

CAPITOLO 5

GRAFICA

In questo capitolo si prenderà in esame come il BASIC gestisce la grafica e si scoprirà che le funzioni grafiche supportate sono di tipo avanzato. Il BASIC è dotato di strumenti grafici sufficienti per poter creare un programma di tipo *paint* relativamente avanzato (in grado di salvare e caricare immagini da e su disco). Si vedrà come si potranno creare cerchi, rettangoli, linee e come attribuire loro i colori. Dopo aver preso in considerazione il programma *paint*, verrà trattata brevemente l'istruzione *DRAW* che consente di produrre alcuni elementi grafici e di applicarvi l'animazione. L'*animazione* è un importante concetto grafico che il BASIC è in grado di gestire consentendo la creazione, lo spostamento e la visualizzazione rapida di immagini grafiche. Nell'esempio viene creato un elemento grafico al quale verrà applicata l'animazione generandone uno spostamento e una visualizzazione su varie parti dello schermo.
Dopo l'animazione verranno presi in considerazione gli strumenti grafici che sono gestiti dal BASIC PDS *Presentation Graphics* per la creazione di grafici a torta, a barre e a linee.

***Nota**: La lettura di questo capitolo potrà risultare molto utile per coloro che dovranno generare grafici di tipo finanziario/commerciale.*

Il capitolo termina presentando un paio di funzioni di basso livello che verificano che tipo di scheda grafica è stata installata nel computer e la risoluzione verticale e orizzontale dello schermo. Questa verifica dell'ambiente grafico consentirà di associare ai programmi tutte le caratteristiche grafiche ad essi necessarie per poter riprodurre graficamente al meglio quanto realizzato.
La grafica, indubbiamente, è un concetto estremamente versatile, come versatile sarà il contenuto di questo capitolo.

UN PROGRAMMA *PAINT* GESTITO DAL MOUSE

Il primo progetto prevede la realizzazione di un programma di tipo *paint* interamente gestito dal mouse. Anche se non si possiede questo tipo di periferica, si tenga presente che il programma che verrà presentato è un veicolo per la comprensione delle varie routine grafiche che vengono supportate dal BASIC, perciò, si consiglia di leggere questo capitolo in qualsiasi caso.
Poiché si dovrà utilizzare il mouse, sarà innanzitutto necessario definire *TYPE RegType*, *InRegs* e *OutRegs* proprio come si è fatto nelle precedenti occasioni in cui si è previsto l'uso del mouse.

```
TYPE RegType
   ax      AS INTEGER
   bx      AS INTEGER
   cx      AS INTEGER
   dx      AS INTEGER
   bp      AS INTEGER
   si      AS INTEGER
   di      AS INTEGER
   flags   AS INTEGER
END TYPE

   DIM InRegs AS RegType, OutRegs AS RegType
   :
   :
```

***Nota**: Un programma paint che non preveda l'uso del mouse è un esercizio inutile poiché l'uso della tastiera per un programma paint non dà alcuna soddisfazione.*

Il programma *paint* che si sta generando dovrà gestire diversi tipi di azioni: il disegno di pixel a video, il disegno di linee, box, cerchi e, infine, il salvataggio/apertura di immagini grafiche presenti su disco. Per questa ragione, è molto importante prima definire un'organizzazione, e sarebbe auspicabile creare il corpo centrale del programma in modo che risulti il più semplice possibile, frammentando il resto del codice in subroutine separate.
Innanzitutto viene creata una struttura di tipo *GOSUB ... RETURN* in modo che la variabili possano essere condivise: si tenga presente che, in questo caso, la creazione di sottoprogrammi non ha alcuna utilità.

```
TYPE RegType
   ax      AS INTEGER
   bx      AS INTEGER
   cx      AS INTEGER
```

```
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

    DIM InRegs AS RegType, OutRegs AS RegType

    GOSUB Initialize
    :
    :
```

> ***Consiglio:*** Nelle precedenti versioni del BASIC, tutte le variabili venivano utilizzate per essere condivise (vale a dire la *variabili globali* e non era possibile utilizzare funzioni o sottoprogrammi ma solo subroutine (per esempio, come quelle di tipo *GOSUB*). Ora, al contrario, vi è la possibilità di utilizzare funzioni e sottoprogrammi e di creare variabili di tipo *locale*. Per poter condividere la variabili sarà quindi necessario utilizzare un'istruzione *SHARED COMMON*.

Nella routine di inizializzazione si potrà impostare la modalità video, visualizzare la barra dei menu e inizializzare il mouse. In effetti, quando il programma viene avviato, la prima operazione prevede l'attesa della pressione del pulsante sinistro del mouse: una selezione da menu potrà essere effettuata premendo il pulsante sinistro del mouse, oppure lo stesso pulsante potrà essere utilizzato per tracciare elementi grafici. È quindi necessario impostare un *loop* perpetuo che rimanga in attesa della pressione del pulsante sinistro:

```
TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

    DIM InRegs AS RegType, OutRegs AS RegType

    GOSUB Initialize

    DO
        GOSUB GetLeftButtonPress
        :
```

```
    :
LOOP WHILE 1

END
```

Si potrà ora impostare una subroutine (*GetLeftButtonPress*) che rimanga in attesa della pressione del pulsante sinistro del mouse. Dopo la pressione, si dovrà sapere se è stata effettuata una selezione da un menu perciò si dovrà impostare *GetLeftButtonPress* in modo che questa imposti una variabile, *MenuSelectionMade%*, a *1*, nel caso in cui la pressione del pulsante è stata effettuata all'interno della barra dei menu, oppure a *0* in caso contrario. Questo consente di controllare l'effettiva selezione di un menu:

```
TYPE RegType
   ax      AS INTEGER
   bx      AS INTEGER
   cx      AS INTEGER
   dx      AS INTEGER
   bp      AS INTEGER
   si      AS INTEGER
   di      AS INTEGER
   flags   AS INTEGER
END TYPE

   DIM InRegs AS RegType, OutRegs AS RegType

   GOSUB Initialize

   DO
      GOSUB GetLeftButtonPress

      IF MenuSelectionMade% THEN
         GOSUB MenuChoice
      :
      :
   LOOP WHILE 1

   END
```

Nel caso in cui sia stata avviata una selezione da menu, allora sarà necessario gestire tale azione in una nuova subroutine alla quale potrà essere assegnato il nome *MenuChoice*. A questo punto sarà necessario impostare i *flag* interni per attivare/disattivare le varie azioni di disegno. Per esempio, se l'utente ha selezionato l'opzione *Linea*, allora sarà necessario impostare il *flag* denominato *LineFlag%*, in *MenuChoice*, a *1*. Avendo selezionato questa opzione si sarà ora pronti a tracciare linee ogni volta che verrà mantenuto premuto il pulsante sinistro del mouse.

Nel caso in cui il pulsante *non sia stato premuto* a livello della prima riga dello schermo, *MenuSelectionMade%* non verrà impostato poiché si suppone che venga

avviata un'operazione di disegno. Si potrà controllare qual è l'azione da avviare verificando il *flag* che è stato impostato in *MenuChoice*.

```
TYPE RegType
   ax      AS INTEGER
   bx      AS INTEGER
   cx      AS INTEGER
   dx      AS INTEGER
   bp      AS INTEGER
   si      AS INTEGER
   di      AS INTEGER
   flags   AS INTEGER
END TYPE

   DIM InRegs AS RegType, OutRegs AS RegType
   DIM Storage(31266) AS INTEGER

   GOSUB Initialize

   DO
      GOSUB GetLeftButtonPress

      IF MenuSelectionMade% THEN
         GOSUB MenuChoice
      ELSE
         IF DrawFlag% THEN GOSUB DrawPixel
         IF LineFlag% THEN GOSUB DrawLine
         IF BoxFlag% THEN GOSUB DrawBox
         IF CircleFlag% THEN GOSUB DrawCircle
         IF PaintFlag% THEN GOSUB DrawPaint
      END IF
   LOOP WHILE 1

   END
```

Consiglio: Come si può notare, la procedura principale *PAINT.BAS* è stata creata utilizzando una tecnica di *modularità*. Il risultato è che questa procedura consente una migliore comprensione del programma e una più facile operazione di *debug*. La divisione dell'intero codice in componenti di questo tipo viene definita *programmazione sequenziale*, o *programmazione top-down*. Questa procedura viene spesso utilizzata nei programmi di tipo professionale.

In altre parole l'azione si svolge secondo questo schema: l'utente potrà iniziare a disegnare linee solo se prima avrà selezionato l'opzione *Linea* dalla barra dei menu. Quando verrà effettuata una selezione di questo tipo, si passerà alla subroutine *MenuChoice* per impostare *LineFlag%* a *1*.

In seguito l'utente rilascia il pulsante sinistro del mouse e il programma esce da *MenuChoice* per riportare il controllo all'inizio del *loop* principale in attesa di una nuova pressione del pulsante sinistro del mouse che verrà gestita da *GetLeftButton-Press*. L'utente, a questo punto, sposterà il puntatore del mouse all'interno dell'area di lavoro, posta al di sotto della barra dei menu, e premerà nuovamente il pulsante sinistro del mouse. Internamente, si uscirà da *GetLeftButtonPress* senza che *Menu-SelectionMade%* sia stato impostato perciò si dovrà controllare se *LineFlag%* è stato impostato e si accederà alla subroutine appropriata (*DrawLine*). La permanenza all'interno di questa subroutine verrà mantenuta fintanto che l'utente non rilascerà il pulsante sinistro del mouse.
Dal punto di vista dell'utente, l'inizio della linea verrà ancorato alla posizione in cui sarà stato premuto il pulsante sinistro del mouse e lo spostamento del puntatore all'interno dell'area di lavoro determinerà la posizione finale della linea. Quando verrà rilasciato il pulsante la linea verrà "fissata" e *DrawLine* rimarrà in attesa di una successiva pressione del pulsante sinistro che genererà la creazione di una nuova linea o la selezione di una nuova opzione dalla barra dei menu.
Questo è uno schema generalizzato dal punto di vista della programmazione. La procedura principale è ora pressoché completa (si noti che è stato inserito un array (*Storage( )*) che sarà in grado di memorizzare il contenuto dell'area di lavoro su disco. A questo punto, non rimane altro che definire le subroutine.

SUBROUTINE *INITIALIZE*

Nella subroutine di inizializzazione si dovrà innanzitutto impostare una modalità video e la visualizzazione della barra dei menu.

```
Initialize:
   SCREEN 8
   LOCATE 1, 1
   Fore% = 1
   Back% = 2
   COLOR Fore%, Back%
   PRINT "Esci    Colore    Sfondo    Draw    Linea    "⌉
   + "Box     Cerchio  Paint    Salva    Apri";
   :
   :
```

Di seguito le opzioni della barra dei menu del programma *paint*:

Opzioni menu	*Descrizione*
Esci	Esce dal programma.
Colore	Imposta il colore di primo piano dell'elemento grafico.
Sfondo	Imposta il colore dello sfondo.

Draw	Disegna un pixel alla volta.
Linea	Traccia una linea.
Box	Traccia un elemento rettangolare chiuso.
Cerchio	Traccia un elemento circolare chiuso.
Paint	Riempie un elemento chiuso con un colore pieno.
Salva	Salva l'immagine su disco.
Apri	Carica un'immagine prelevandola da disco.

Sarà ora necessario inizializzare il mouse e visualizzarne il puntatore.

```
Initialize:
   SCREEN 8
   LOCATE 1, 1
   Fore% = 1
   Back% = 2
   COLOR Fore%, Back%
   PRINT "Esci    Colore    Sfondo    Draw    Linea    "⌉
   + "Box     Cerchio  Paint    Salva    Apri";
   InRegs.ax = 0                          'Inizializza mouse
   CALL INTERRUPT(&H33, InRegs, OutRegs)
   InRegs.ax = 1                          'Mostra puntatore mouse
   CALL INTERRUPT(&H33, InRegs, OutRegs)
```

Infine, per terminare la procedura, si potranno impostare tutti i *flag* a *0*.

```
Initialize:
   SCREEN 8
   LOCATE 1, 1
   Fore% = 1
   Back% = 2
   COLOR Fore%, Back%
   PRINT "Esci    Colore    Sfondo    Draw    Linea    "⌉
   + "Box     Cerchio  Paint    Salva    Apri";
   InRegs.ax = 0                          'Inizializza mouse
   CALL INTERRUPT(&H33, InRegs, OutRegs)
   InRegs.ax = 1                          'Mostra puntatore mouse
   CALL INTERRUPT(&H33, InRegs, OutRegs)
   DrawFlag% = 0
   LineFlag% = 0
   BoxFlag% = 0
   CircleFlag% = 0
   PaintFlag% = 0
   RETURN
```

Si è ora pronti a creare la subroutine *GetLeftButtonPress*.

LA SUBROUTINE *GetLeftButtonPress*

La creazione di questa subroutine è relativamente semplice: in sostanza quello che ad essa viene richiesto è di attendere la pressione del pulsante sinistro del mouse, azione questa che viene controllata con l'interrupt &H33, servizio 5.

```
GetLeftButtonPress:

   DO
      InRegs.bx = 0          'Attende pressione pulsante sinistro
      InRegs.ax = 5
      CALL INTERRUPT(&H33, InRegs, OutRegs)
   LOOP WHILE OutRegs.bx = 0
   :
   :
```

Ora si dovrà impostare la variabile *menuSelectionMade%* in modo che questa indichi se è stata o meno avviata un'azione di selezione dalla barra dei menu. Questo controllo potrà essere effettuato semplicemente verificando che il puntatore del mouse sia posizionato sulla riga *1*, corrispondente alla riga contenente la barra dei menu.

```
GetLeftButtonPress:

   DO
      InRegs.bx = 0          'Attende pressione pulsante sinistro
      InRegs.ax = 5
      CALL INTERRUPT(&H33, InRegs, OutRegs)
   LOOP WHILE OutRegs.bx = 0
      Row% = OutRegs.dx \ 8 + 1
      IF Row% = 1 THEN
         MenuSelectionMade% = 1
      ELSE
         MenuSelectionMade% = 0
      END IF
      RETURN
```

Terminata la creazione di questa subroutine, si sarà ora pronti a gestire le selezioni dalla barra dei menu.

LA SUBROUTINE *MenuChoice*

Nella routine *MenuChoice* si potrà conservare traccia di quale selezione è stata effettuata dalla barra dei menu. La barra dei menu dovrebbe apparire simile a quella riportata in figura 5.1.

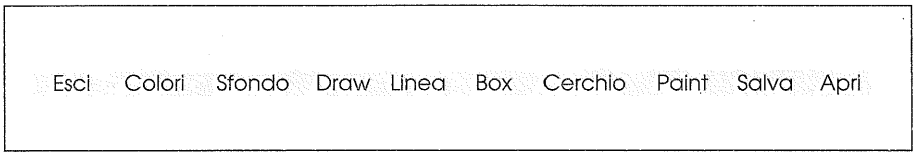

Figura 5.1

Si analizzerà ora un'opzione alla volta. Innanzitutto sarà necessario reimpostare i *flag* di disegno e verificare quale selezione è stata effettuata. Tenere presente che ogni voce della barra dei menu occupa otto colonne e ogni colonna è composta da otto pixel.

```
MenuChoice:
      DrawFlag% = 0
      LineFlag% = 0
      BoxFlag% = 0
      CircleFlag% = 0
      PaintFlag% = 0
      Choice% = OutRegs.cx \ 64 + 1
      :
      :
```

Poi, si potrà avviare quanto richiesto dalla selezione effettuata servendosi di un'istruzione *SELECT CASE*. Per esempio, per la selezione *1*, corrispondente a *Esci*, sarà sufficiente chiudere il programma.

```
MenuChoice:
      DrawFlag% = 0
      LineFlag% = 0
      BoxFlag% = 0
      CircleFlag% = 0
      PaintFlag% = 0
      Choice% = OutRegs.cx \ 64 + 1
      SELECT CASE Choice%
      CASE 1
         END
         :
         :
```

La selezione *2* consente di impostare il colore di primo piano e ogni volta che l'utente cliccherà su questa selezione il colore di primo piano verrà incrementato ciclicamente di un valore all'interno dei 16 colori disponibili in questa modalità. I colori selezionati potranno essere rappresentati dalla visualizzazione della parola *Colore* alla quale viene associato il colore attivato. Si noti che il puntatore del mouse viene nascosto o visualizzato prima di operare sullo schermo.

```
MenuChoice:
      DrawFlag% = 0
      LineFlag% = 0
      BoxFlag% = 0
      CircleFlag% = 0
      PaintFlag% = 0
      Choice% = OutRegs.cx \ 64 + 1
      SELECT CASE Choice%
      CASE 1
         END
      CASE 2
         Fore% = Fore% + 1
         IF Fore% > 15 THEN Fore% = 0
         COLOR Fore%, Back%
         InRegs.ax = 2                    'Nasconde puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         LOCATE 1, 9
         PRINT "Colore   ";
         InRegs.ax = 1                    'Visualizza puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         :
         :
```

La successiva opzione modifica il colore di sfondo (operazione gestita dall'istruzione *COLOR* del BASIC). Ogni volta che verrà modificato il colore dello sfondo, questo verrà riflesso a video, l'utente perciò potrà effettuare la sua scelta molto semplicemente.

```
MenuChoice:
      DrawFlag% = 0
      LineFlag% = 0
      BoxFlag% = 0
      CircleFlag% = 0
      PaintFlag% = 0
      Choice% = OutRegs.cx \ 64 + 1
      SELECT CASE Choice%
      CASE 1
         END
      CASE 2
         Fore% = Fore% + 1
         IF Fore% > 15 THEN Fore% = 0
```

```
      COLOR Fore%, Back%
      InRegs.ax = 2                    'Nasconde puntatore mouse
      CALL INTERRUPT(&H33, InRegs, OutRegs)
      LOCATE 1, 9
      PRINT "Colore    ";
      InRegs.ax = 1                    'Visualizza puntatore mouse
      CALL INTERRUPT(&H33, InRegs, OutRegs)
   CASE 3
      Back% = Back% + 1
      IF Back% > 7 THEN Back% = 0
      COLOR Fore%, Back%
      :
      :
```

Si passerà ora alla definizione della altre opzioni della barra dei menu (figura 5.2). Ognuna di queste opzioni dovrà essere gestita con proprie subroutine ma qui si dovrà ora impostarne i *flag* appropriati.

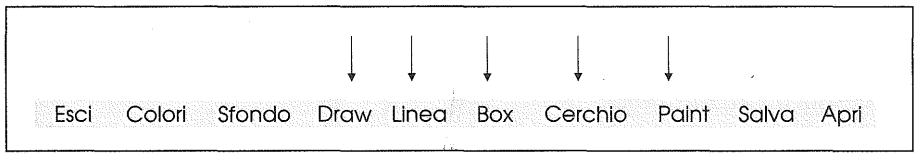

Figura 5.2

```
MenuChoice:
      DrawFlag% = 0
      LineFlag% = 0
      BoxFlag% = 0
      CircleFlag% = 0
      PaintFlag% = 0
      Choice% = OutRegs.cx \ 64 + 1
      SELECT CASE Choice%
      CASE 1
         END
      CASE 2
         Fore% = Fore% + 1
         IF Fore% > 15 THEN Fore% = 0
         COLOR Fore%, Back%
         InRegs.ax = 2                 'Nasconde puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         LOCATE 1, 9
         PRINT "Colore    ";
         InRegs.ax = 1                 'Visualizza puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
```

```
CASE 3
   Back% = Back% + 1
   IF Back% > 7 THEN Back% = 0
   COLOR Fore%, Back%
CASE 4
   DrawFlag% = 1
CASE 5
   LineFlag% = 1
CASE 6
   BoxFlag% = 1
CASE 7
   CircleFlag% = 1
CASE 8
   PaintFlag% = 1
   :
   :
```

La successiva opzione, *Salva*, consente di salvare l'immagine presente a video. In particolare, essa potrà essere salvata in un file a cui verrà assegnato il nome *PAINT.DAT*. Poiché la selezione dell'opzione genera il salvataggio del file, essa dovrà essere gestita da *MenuChoice*.

***Nota**: Sarebbe più professionale consentire all'utente di selezionare il nome del file da salvare su disco.*

Per salvare l'immagine a video, viene utilizzata l'istruzione *GET* del BASIC che la carica in un array. Sarà quindi necessario prima impostare l'array di valori interi al quale assegnare il nome, per esempio, *Storage( )*. Dato che si desidera salvare solo l'area di lavoro, sarà necessario tenere presente che l'angolo superiore sinistro corrisponde a *(X1, Y1) = (0, 8)* (viene esclusa la barra dei menu) e l'angolo inferiore destro corrisponde a *(X2, Y2) = (639, 199)*. Il numero di byte necessario per memorizzare l'immagine sarà dato dalla formula:

$$Byte = 4 + INT(((X2-X1+1)*(Bit/Pixel/Piano)+7)/8*Piani*((Y2-Y1)+1)$$

dove: *Bit/Pixel/Piano* corrisponde al numero di bit per pixel per piano e *Piani* è il numero di piani gestiti dalla modalità video attiva. Per determinare questi valori per una modalità video specifica si veda la documentazione BASIC (o analizzare la subroutine *SaveSprite* che viene riportata più avanti in questo capitolo). Per la modalità video 8 e data la dimensione dell'immagine da salvare, saranno necessari 62.535 byte oppure 31.266 valori interi.
A questo punto, l'istruzione *GET* dovrà essere immessa come di seguito riportato:

```
GET (0, 8)-(639, 199),Storage
```

Questa invocazione caricherà l'immagine a video nell'array *Storage( )* (si noti che sarà necessario prima disattivare la visualizzazione del puntatore del mouse: in caso contrario, verrà anch'esso salvato insieme all'immagine).
Successivamente si potrà trasferire il contenuto dell'array *Storage( )* direttamente al file su disco utilizzando la routine di BASIC di salvataggio in formato binario *BSAVE*. Per poter eseguire questa operazione, sarà necessario assegnare a *BSAVE* l'indirizzo dei dati che si intendono salvare e il numero dei byte che dovranno essere salvati.

> ***Consiglio***: L'istruzione *BSAVE* del BASIC è straordinariamente potente: è questo il sistema più rapido gestito da BASIC per caricare o trasferire dati da/a disco. Sebbene la si utilizzi in questa occasione per salvare un'immagine grafica, questa istruzione viene utilizzata più frequentemente per salvare velocemente blocchi di dati, specialmente per quelli contenuti negli array.

Come si è già visto, gli indirizzi di memoria sono composti da due word, un indirizzo di *segmento* e un indirizzo di *offset*. In breve, un *segmento* corrisponde a 64 Kb di memoria e un *offset* corrisponde all'allocazione di un particolare byte all'interno del segmento, a partire dall'inizio del segmento stesso (figura 5.3).

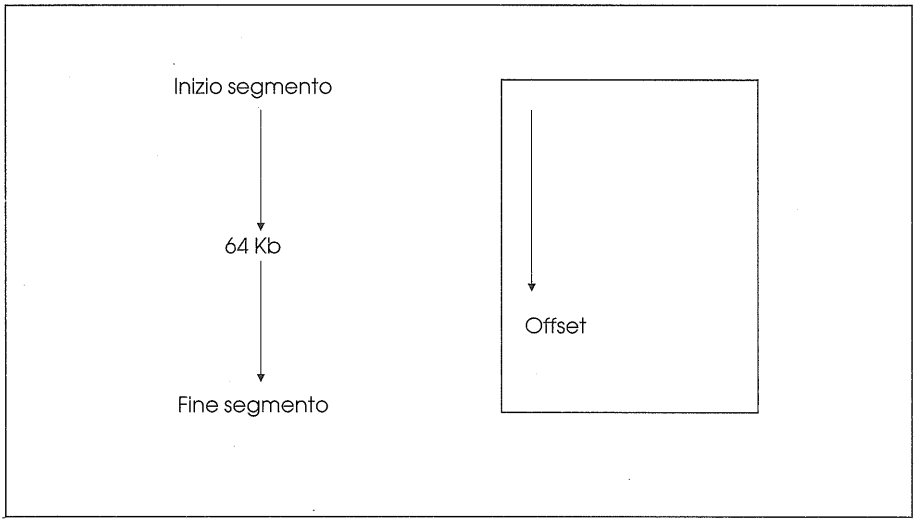

Figura 5.3

I segmenti e gli offset verranno trattati più dettagliatamente nel capitolo 10: per ora basti sapere che *BSAVE* dovrà essere utilizzato sia per gli indirizzi del segmento che per quelli dell'offset dell'array *Storage( )*. Per ottenere l'indirizzo del segmento, il

BASIC prevede l'uso della funzione *VARSEG( )*, mentre per ottenere l'indirizzo dell'offset il BASIC prevede l'uso della funzione *VARPTR( )*.
Si inizierà quindi con l'impostazione dell'indirizzo del segmento che il BASIC renderà uguale all'indirizzo del segmento contenuto in *Storage( )*:

```
DEF SEG = VARSEG(Storage(1))
```

dove *Storage(1)* corrisponde al primo elemento dell'array. Successivamente sarà necessario trasferire il nome del file che si intende memorizzare su disco (per esempio, *PAINT.DAT*), seguito dall'indirizzo dell'offset del primo byte da scrivere (corrispondente a *VARPTR(1)*) e dal numero di byte da scrivere: 62.532.

```
DEF SEG = VARSEG(Storage(1))
BSAVE "PAINT.DAT", VARPTR(Storage(1)), 62532
DEF SEG
```

Al termine verrà reimpostato l'indirizzo del segmento che il BASIC utilizzerà per riportare i dati al loro valore originale servendosi di DEF SEG (nessun argomento).

```
MenuChoice:
    DrawFlag% = 0
    LineFlag% = 0
    BoxFlag% = 0
    CircleFlag% = 0
    PaintFlag% = 0
    Choice% = OutRegs.cx \ 64 + 1
    SELECT CASE Choice%
    CASE 1
       END
    CASE 2
       Fore% = Fore% + 1
       IF Fore% > 15 THEN Fore% = 0
       COLOR Fore%, Back%
       InRegs.ax = 2                 'Nasconde puntatore mouse
       CALL INTERRUPT(&H33, InRegs, OutRegs)
       LOCATE 1, 9
       PRINT "Colore   ";
       InRegs.ax = 1                 'Visualizza puntatore mouse
       CALL INTERRUPT(&H33, InRegs, OutRegs)
    CASE 3
       Back% = Back% + 1
       IF Back% > 7 THEN Back% = 0
       COLOR Fore%, Back%
    CASE 4
       DrawFlag% = 1
    CASE 5
       LineFlag% = 1
    CASE 6
       BoxFlag% = 1
```

```
CASE 7
   CircleFlag% = 1
CASE 8
   PaintFlag% = 1
CASE 9
   InRegs.ax = 2                    'Nasconde puntatore mouse
   CALL INTERRUPT(&H33, InRegs, OutRegs)
   GET (0, 8)-(639, 199), Storage
   DEF SEG = VARSEG(Storage(1))
   BSAVE "PAINT.DAT", VARPTR(Storage(1)), 62532
   DEF SEG
   InRegs.ax = 1                    'Visualizza puntatore mouse
   CALL INTERRUPT(&H33, InRegs, OutRegs)
   :
   :
```

A questo punto si sarà in grado di utilizzare *BSAVE* per salvare l'immagine grafica su disco.
L'ultima opzione, *Apri*, consente di leggere l'immagine da disco e portarla a video. Questa operazione potrà essere effettuata con *BLOAD*, funzione di BASIC in grado di caricare un formato binario e con *PUT*. Ancora una volta, sarà necessario indicare a *BLOAD* l'indirizzo dell'inizio dell'array *Storage( )*.

> ***Consiglio***: Proprio come *BSAVE* risulta essere il sistema più rapido a disposizione del BASIC per salvare i dati su disco, *BLOAD* è il sistema più rapido per caricarli in memoria.

A questo punto sarà necessario nascondere il puntatore del mouse e caricare *PAINT.DAT* in memoria:

```
MenuChoice:
      DrawFlag% = 0
      LineFlag% = 0
      BoxFlag% = 0
      CircleFlag% = 0
      PaintFlag% = 0
      Choice% = OutRegs.cx \ 64 + 1
      SELECT CASE Choice%
      CASE 1
         END
      CASE 2
         Fore% = Fore% + 1
         IF Fore% > 15 THEN Fore% = 0
         COLOR Fore%, Back%
         InRegs.ax = 2                    'Nasconde puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
```

```
      LOCATE 1, 9
      PRINT "Colore    ";
      InRegs.ax = 1                    'Visualizza puntatore mouse
      CALL INTERRUPT(&H33, InRegs, OutRegs)
   CASE 3
      Back% = Back% + 1
      IF Back% > 7 THEN Back% = 0
      COLOR Fore%, Back%
   CASE 4
      DrawFlag% = 1
   CASE 5
      LineFlag% = 1
   CASE 6
      BoxFlag% = 1
   CASE 7
      CircleFlag% = 1
   CASE 8
      PaintFlag% = 1
   CASE 9
      InRegs.ax = 2                    'Nasconde puntatore mouse
      CALL INTERRUPT(&H33, InRegs, OutRegs)
      GET (0, 8)-(639, 199), Storage
      DEF SEG = VARSEG(Storage(1))
      BSAVE "PAINT.DAT", VARPTR(Storage(1)), 62532
      DEF SEG
      InRegs.ax = 1                    'Visualizza puntatore mouse
      CALL INTERRUPT(&H33, InRegs, OutRegs)
   CASE 10
      InRegs.ax = 2                    'Nasconde puntatore mouse
      CALL INTERRUPT(&H33, InRegs, OutRegs)
      DEF SEG = VARSEG(Storage(1))
      BLOAD "PAINT.DAT", VARPTR(Storage(1))
      :
      :
```

Successivamente, sarà possibile riportare l'immagine a video utilizzando l'istruzione *PUT* del BASIC nella forma *PUT(0, 8), Storage, PSET*. Le istruzioni *GET* e *PUT* trasferiscono rapidamente i dati da e a disco e ambedue sono due pietre miliari delle routine grafiche del BASIC. L'opzione *PSET* indica, in questo caso, che ogni bit che viene immesso a video dovrà sovrascrivere quanto già presente nella medesima posizione.

Per default, se non viene specificata alcuna opzione del tipo *PSET*, viene assunto *XOR* che va a sostituire qualsiasi cosa sia presente a video. Questa opzione verrà dettagliatamente descritta quando si lavorerà sulle operazioni di gestione del video.

Consiglio: *XOR* ha una speciale proprietà: se *A* viene associato a *B* tramite *XOR* il risultato sarà comunque *B* mentre *A* rimarrà in secondo piano. Questa è una ragione per cui il puntatore del mouse viene spesso gestito tramite *XOR* con i caratteri a video. Quando il carattere a video verrà anch'esso associato al puntatore tramite *XOR*, verrà nuovamente visualizzato il carattere originale e il puntatore potrà essere spostato altrove.

Infine, sarà necessario ora ripristinare la visualizzazione del puntatore del mouse. Di seguito viene riportata l'intera subroutine *MenuChoice*.

```
MenuChoice:
     DrawFlag% = 0
     LineFlag% = 0
     BoxFlag% = 0
     CircleFlag% = 0
     PaintFlag% = 0
     Choice% = OutRegs.cx \ 64 + 1
     SELECT CASE Choice%
     CASE 1
        END
     CASE 2
        Fore% = Fore% + 1
        IF Fore% > 15 THEN Fore% = 0
        COLOR Fore%, Back%
        InRegs.ax = 2                 'Nasconde puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        LOCATE 1, 9
        PRINT "Colore   ";
        InRegs.ax = 1                 'Visualizza puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
     CASE 3
        Back% = Back% + 1
        IF Back% > 7 THEN Back% = 0
        COLOR Fore%, Back%
     CASE 4
        DrawFlag% = 1
     CASE 5
        LineFlag% = 1
     CASE 6
        BoxFlag% = 1
     CASE 7
        CircleFlag% = 1
     CASE 8
        PaintFlag% = 1
     CASE 9
        InRegs.ax = 2                 'Nasconde puntatore mouse
```

```
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        GET (0, 8)-(639, 199), Storage
        DEF SEG = VARSEG(Storage(1))
        BSAVE "PAINT.DAT", VARPTR(Storage(1)), 62532
        DEF SEG
        InRegs.ax = 1                 'Visualizza puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
     CASE 10
        InRegs.ax = 2                 'Nasconde puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        DEF SEG = VARSEG(Storage(1))
        BLOAD "PAINT.DAT", VARPTR(Storage(1))
        DEF SEG
        PUT (0, 8), Storage, PSET
        InRegs.ax = 1                 'Visualizza puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
     END SELECT
  RETURN
```

LA SUBROUTINE *DrawPixel*

La procedura principale, fino ad ora ha gestito tutte le operazioni necessarie, tranne quelle di disegno.

```
TYPE RegType
   ax      AS INTEGER
   bx      AS INTEGER
   cx      AS INTEGER
   dx      AS INTEGER
   bp      AS INTEGER
   si      AS INTEGER
   di      AS INTEGER
   flags   AS INTEGER
END TYPE

   DIM InRegs AS RegType, OutRegs AS RegType
   DIM Storage(31266) AS INTEGER

   GOSUB Initialize

   DO
      GOSUB GetLeftButtonPress

      IF MenuSelectionMade% THEN
         GOSUB MenuChoice
      ELSE
         IF DrawFlag% THEN GOSUB DrawPixel
```

```
        IF LineFlag% THEN GOSUB DrawLine
        IF BoxFlag% THEN GOSUB DrawBox
        IF CircleFlag% THEN GOSUB DrawCircle
        IF PaintFlag% THEN GOSUB DrawPaint
      END IF
    LOOP WHILE 1

    END
    :
    :
```

Ora è arrivato il momento di utilizzarle. La prima (*DrawPixel*), quando viene richiamata, imposta il pixel alla posizione in cui è posizionato il puntatore del mouse fino a quando non ne verrà rilasciato il pulsante. In questo modo l'utente avrà la possibilità di disegnare un'immagine premendo il pulsante sinistro del mouse per attivare un pixel e spostare il mouse all'interno dell'area di lavoro.

Si inizierà con lo svuotamento del contenuto della coda relativa al rilascio del pulsante del mouse in modo che non venga preso in considerazione la precedente azione collegata a un precedente rilascio del pulsante del mouse stesso.

```
DrawPixel:
     InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
     InRegs.ax = 6
     CALL INTERRUPT(&H33, InRegs, OutRegs)
     :
     :
```

Ora sarà necessario creare un *loop* fino a quando non verrà rilasciato il pulsante del mouse (controllando l'interrupt &H33, servizio 6).

```
DrawPixel:
     InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
     InRegs.ax = 6
     CALL INTERRUPT(&H33, InRegs, OutRegs)

     DO
        :
        [Imposta il pixel attivo]
        :
        InRegs.bx = 0              'Rilascio pulsante sinistro
        InRegs.ax = 6
        CALL INTERRUPT(&H33, InRegs, OutRegs)
     LOOP WHILE OutRegs.bx = 0

   RETURN
```

Ogni volta che viene avviato questo e il programma rimarrà in attesa del rilascio del pulsante del mouse, sarà necessario indicare la posizione del puntatore e impostarne il pixel assegnandoli l'appropriato colore di primo piano. La posizione corrente del

puntatore potrà essere ottenuta con l'interrupt &H33, servizio 3, si dovrà nascondere il puntatore del mouse con il servizio 2 e si imposterà il pixel con l'istruzione *PSET( )* del BASIC. Infine dovrà essere nuovamente visualizzato il puntatore:

```
DrawPixel:
     InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
     InRegs.ax = 6
     CALL INTERRUPT(&H33, InRegs, OutRegs)

     DO
        InRegs.ax = 3
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        X% = OutRegs.cx
        Y% = OutRegs.dx
        InRegs.ax = 2                  'Nasconde puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        PSET (X%, Y%)
        InRegs.ax = 1                  'Visualizza puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        InRegs.bx = 0                  'Rilascio pulsante sinistro
        InRegs.ax = 6
        CALL INTERRUPT(&H33, InRegs, OutRegs)
     LOOP WHILE OutRegs.bx = 0

   RETURN
```

Tutto il lavoro viene in sostanza svolto da *PSET(X%, Y%)* che imposta il pixel alla posizione occupata dal puntatore associandogli il colore di primo piano. E con questo si è terminata la realizzazione della prima subroutine di disegno.

LA SUBROUTINE *DrawLine*

La successiva subroutine è *DrawLine*.

```
TYPE RegType
   ax      AS INTEGER
   bx      AS INTEGER
   cx      AS INTEGER
   dx      AS INTEGER
   bp      AS INTEGER
   si      AS INTEGER
   di      AS INTEGER
   flags   AS INTEGER
END TYPE

   DIM InRegs AS RegType, OutRegs AS RegType
   DIM Storage(31266) AS INTEGER
```

```
GOSUB Initialize

DO
   GOSUB GetLeftButtonPress

   IF MenuSelectionMade% THEN
      GOSUB MenuChoice
   ELSE
      IF DrawFlag% THEN GOSUB DrawPixel
      IF LineFlag% THEN GOSUB DrawLine
      IF BoxFlag% THEN GOSUB DrawBox
      IF CircleFlag% THEN GOSUB DrawCircle
      IF PaintFlag% THEN GOSUB DrawPaint
   END IF
LOOP WHILE 1

END
```

Questa operazione potrà essere gestita con l'istruzione *LINE* del BASIC che consente di tracciare una linea tra due punti definiti dello schermo *(X1, Y1)* e *(X2, Y2)*.

```
LINE (X1, Y1) - (X2, Y2), Color%
```

In questo caso, *Color%* è un argomento facoltativo che specifica il colore che verrà applicato alla riga. Questo argomento verrà utilizzato sia per tracciare che per cancellare linee. Quando si arriverà alla subroutine *DrawLine* l'utente in effetti avrà già cliccato con il mouse sull'area di lavoro per stabilire il punto di "ancoraggio" di partenza della linea. Poiché le variabili sono globali (ovvero condivise da tutte le subroutine), la posizione saranno contenute in *OutRegs.cx* e *OutRegs.dx* che sono ancora collegate alla posizione sulla quale l'utente ha premuto il pulsante del mouse (memorizzata in *GetLeftButtonPress*).

```
DrawLine:
      X% = OutRegs.cx
      Y% = OutRegs.dx
      :
      :
```

Quanto sopra indica il punto di ancoraggio della linea, *(X%, Y%)* e quando l'utente sposterà il puntatore nella nuova posizione, *(XNew%, YNew%)* la linea verrà tracciata da *(X%, Y%)* a *(XNew%, YNew%)* (figura 5.4).

Quando invece il puntatore verrà spostato a una nuova posizione, *XNew%, YNew%* verranno trasposti a *XOld%* e *YOld%*: la cancellazione della linea potrà essere effettuata assegnandole lo stesso colore dello sfondo (Figura 5.5).

Sarà ora possibile tracciare una nuova riga (*XNew%, YNew%*) (figura 5.6).

In questo modo la parte terminale della linea verrà sempre ancorata alla posizione su cui si è cliccato mentre l'altra seguirà il movimento del puntatore. Quando verrà

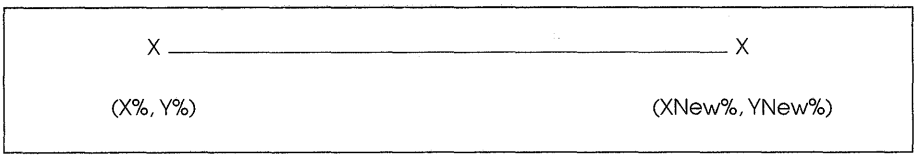

Figura 5.4

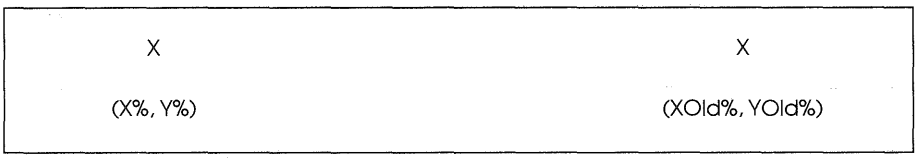

Figura 5.5

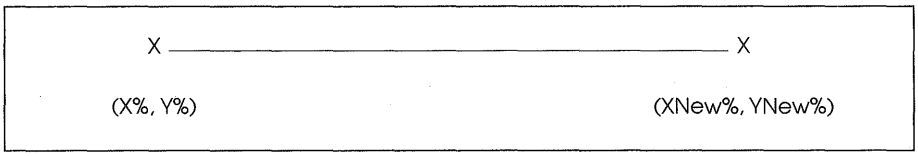

Figura 5.6

rilasciato il pulsante sinistro del mouse la linea risulterà permanentemente "ancorata" tra i suoi estremi. Per eseguire quest'ultima operazione sarà necessario svuotare la coda delle operazioni legate al rilascio del pulsante sinistro del mouse avviando un *loop* che rimane in attesa del rilascio del pulsante.

```
DrawLine:
     X% = OutRegs.cx
     Y% = OutRegs.dx
     XOld% = X%
     YOld% = Y%
     InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
     InRegs.ax = 6
     CALL INTERRUPT(&H33, InRegs, OutRegs)
     DO
        :
        :
        InRegs.bx = 0              'Rilascio pulsante sinistro
        InRegs.ax = 6
        CALL INTERRUPT(&H33, InRegs, OutRegs)
     LOOP WHILE OutRegs.bx = 0
  RETURN
```

Ogni volta che verrà avviato il *loop* sarà necessario controllare la nuova posizione del puntatore cancellando la precedente linea e tracciandone una nuova (dopo aver nascosto il puntatore del mouse):

```
DrawLine:
     X% = OutRegs.cx
     Y% = OutRegs.dx
     XOld% = X%
     YOld% = Y%

     InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
     InRegs.ax = 6
     CALL INTERRUPT(&H33, InRegs, OutRegs)

     DO
        InRegs.ax = 3
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        YNew% = OutRegs.dx
        XNew% = OutRegs.cx
        InRegs.ax = 2                 'Nasconde puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        LINE (X%, Y%)-(XOld%, YOld%), Back%
        LINE (X%, Y%)-(XNew%, YNew%), Fore%
        XOld% = XNew%
        YOld% = YNew%
        InRegs.ax = 1                 'Visualizza puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        InRegs.bx = 0              'Rilascio pulsante sinistro
        InRegs.ax = 6
        CALL INTERRUPT(&H33, InRegs, OutRegs)
     LOOP WHILE OutRegs.bx = 0
   RETURN
```

Quando verrà rilasciato il pulsante del mouse la linea diventerà "permanente".

LA SUBROUTINE *DrawBox*

La routine successiva (*DrawBox*) dopo aver esaminato la precedente routine per il tracciamento delle linee, è molto semplice in quanto, in sostanza, si tratta unicamente di aggiungere il parametro *B* alla fine dell'istruzione *LINE* del BASIC.

```
LINE (x1, y1) - (x2, y2), cOLOR%, B
```

> ***Consiglio***: Se dopo l'opzione *B* viene aggiunta anche l'opzione *F*, si indicherà al BASIC di riempire il box che viene creato con lo stesso colore utilizzato per il contorni.

Si otterrà così un box che si espande dall'angolo superiore sinistro (*X1, Y1*) all'angolo inferiore destro (*X2, Y2*). La semplice aggiunta del parametro *B* al *DrawLine* significa che il punto di ancoraggio ora viene identificato con un'angolo del box e il puntatore del mouse potrà essere spostato all'interno dell'area di lavoro. Quando verrà rilasciato il pulsante del mouse il box verrà reso "permanente".

```
DrawBox:
     X% = OutRegs.cx
     Y% = OutRegs.dx
     XOld% = X%
     YOld% = Y%

     InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
     InRegs.ax = 6
     CALL INTERRUPT(&H33, InRegs, OutRegs)

     DO
        InRegs.ax = 3
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        YNew% = OutRegs.dx
        XNew% = OutRegs.cx
        InRegs.ax = 2                 'Nasconde puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        LINE (X%, Y%)-(XOld%, YOld%), Back%, B
        LINE (X%, Y%)-(XNew%, YNew%), Fore%, B
        XOld% = XNew%
        YOld% = YNew%
        InRegs.ax = 1                 'Visualizza puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        InRegs.bx = 0                 'Rilascio pulsante sinistro
        InRegs.ax = 6
        CALL INTERRUPT(&H33, InRegs, OutRegs)
     LOOP WHILE OutRegs.bx = 0
  RETURN
```

LA SUBROUTINE *DrawCircle*

Gli elementi circolari, in BASIC, potranno anche essere realizzati con l'istruzione *CIRCLE*.

CIRCLE (X%, Y%), Radius!, Color%

> ***Consiglio***: Il BASIC consente anche di utilizzare l'istruzione *CIRCLE* per tracciare archi o ellissi servendosi della seguente istruzione: *CIRCLE (x, y), Radius%, Color%, START!, END!, ASPECT!*.

START! e *END!* indicano, rispettivamente, gli angoli iniziale e finale di un arco (espresso in radianti). *ASPECT!* imposta, invece, il rapporto per la creazione di un ellisse.

Nell'esempio riportato verrà tracciato un cerchio centrato a *(X%, Y%)* con raggio espresso da *Radius!* e con colore specificato dal valore contenuto in *Color%*. Anche in questo caso si potrà adattare *DrawLine* "ancorando" il centro del cerchio e spostando il puntatore nell'area di lavoro fino a raggiungere la dimensione desiderata del cerchio (per esempio, il raggio potrà estendersi da *(X%, Y%)* fino a *(XNew%, YNew%)*), Questa operazione potrà essere effettuata modificando semplicemente le istruzioni *LINE* e sostituendole con altrettante istruzioni *CIRCLE*.

```
DrawCircle:
     X% = OutRegs.cx
     Y% = OutRegs.dx
     XOld% = X%
     YOld% = Y%

     InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
     InRegs.ax = 6
     CALL INTERRUPT(&H33, InRegs, OutRegs)

     DO
        InRegs.ax = 3
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        YNew% = OutRegs.dx
        XNew% = OutRegs.cx
        InRegs.ax = 2                    'Nasconde puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        CIRCLE (X%, Y%), SQR((X% - XOld%) ^ 2 + (Y% - YOld%) ^ 2),
             Back%
        CIRCLE (X%, Y%), SQR((X% - XNew%) ^ 2 + (Y% - YNew%) ^ 2)
        XOld% = XNew%
        YOld% = YNew%
        InRegs.ax = 1                    'Visualizza puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        InRegs.bx = 0                    'Rilascio pulsante sinistro
        InRegs.ax = 6
        CALL INTERRUPT(&H33, InRegs, OutRegs)
     LOOP WHILE OutRegs.bx = 0
  RETURN
```

LA SUBROUTINE *DrawPaint*

L'ultima subroutine che verrà sviluppata è *DrawPaint*. Questa subroutine consentirà di riempire gli elementi grafici chiusi con un colore pieno. In questo caso, verrà utilizzata l'istruzione *PAINT* del BASIC:

```
PAINT (X%, Y%)
```

Questa istruzione riempie un area con il colore di primo piano attivo tenendo presente che l'unica limitazione sta nel fatto che l'area deve essere un elemento grafico chiuso il cui bordo potrà avere lo stesso colore di sfondo del riempimento o essere racchiuso da un bordo di colore diverso che potrà essere specificato come di seguito riportato:

```
PAINT (X%, Y%), BorderColor%
```

Questo potrebbe essere un problema poiché il programma *paint* che si sta creando non è così sofisticato da riconoscere, al momento del riempimento, se il bordo che racchiude l'immagine grafica ha o meno lo stesso colore di quello che verrà utilizzato per il riempimento. Sarà necessario, quindi, impostare un colore di primo piano, il che significa che sarà possibile creare immagini che verranno realizzate con il colore di primo piano attivo al momento dell'operazione.
Tenendo presente questa restrizione, un elemento grafico chiuso potrà essere colorato come di seguito riportato:

```
DrawPaint:
     X% = OutRegs.cx
     Y% = OutRegs.dx
     InRegs.ax = 2                    'Nasconde puntatore mouse
     CALL INTERRUPT(&H33, InRegs, OutRegs)
     PAINT (X%, Y%)
     InRegs.ax = 1                    'Visualizza puntatore mouse
     CALL INTERRUPT(&H33, InRegs, OutRegs)
   RETURN
```

IL LISTATO COMPLETO DEL PROGRAMMA *PAINT*

Il listato 5.1 mostra il listato completo del programma *Paint*.

```
TYPE RegType
    ax       AS INTEGER
    bx       AS INTEGER
    cx       AS INTEGER
    dx       AS INTEGER
```

continua

Listato 5.1 *Il programma Paint con gestione del mouse*

```
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

    DIM InRegs AS RegType, OutRegs AS RegType
    DIM Storage(31266) AS INTEGER

    GOSUB Initialize

    DO
       GOSUB GetLeftButtonPress

       IF MenuSelectionMade% THEN
          GOSUB MenuChoice
       ELSE
          IF DrawFlag% THEN GOSUB DrawPixel
          IF LineFlag% THEN GOSUB DrawLine
          IF BoxFlag% THEN GOSUB DrawBox
          IF CircleFlag% THEN GOSUB DrawCircle
          IF PaintFlag% THEN GOSUB DrawPaint
       END IF
    LOOP WHILE 1

    END

Initialize:
    SCREEN 8
    LOCATE 1, 1
    Fore% = 1
    Back% = 2
    COLOR Fore%, Back%
    PRINT "Esci    Colore   Sfondo   Draw    Linea   "
          + "Box     Cerchio  Paint   Salva    Apri";

    InRegs.ax = 0                          'Inizializza mouse
    CALL INTERRUPT(&H33, InRegs, OutRegs)

    InRegs.ax = 1                          'Mostra puntatore mouse
    CALL INTERRUPT(&H33, InRegs, OutRegs)
    DrawFlag% = 0
    LineFlag% = 0
    BoxFlag% = 0
    CircleFlag% = 0
```

continua

Listato 5.1 *Il programma Paint con gestione del mouse*

```
    PaintFlag% = 0
    RETURN

GetLeftButtonPress:

    DO
       InRegs.bx = 0          'Attende pressione pulsante sinistro
       InRegs.ax = 5
       CALL INTERRUPT(&H33, InRegs, OutRegs)
    LOOP WHILE OutRegs.bx = 0
       Row% = OutRegs.dx \ 8 + 1
       IF Row% = 1 THEN
          MenuSelectionMade% = 1
       ELSE
          MenuSelectionMade% = 0
       END IF
       RETURN

MenuChoice:
       DrawFlag% = 0
       LineFlag% = 0
       BoxFlag% = 0
       CircleFlag% = 0
       PaintFlag% = 0
       Choice% = OutRegs.cx \ 64 + 1
       SELECT CASE Choice%
       CASE 1
          END
       CASE 2
          Fore% = Fore% + 1
          IF Fore% > 15 THEN Fore% = 0
          COLOR Fore%, Back%
          InRegs.ax = 2                 'Nasconde puntatore mouse
          CALL INTERRUPT(&H33, InRegs, OutRegs)
          LOCATE 1, 9
          PRINT "Colore   ";
          InRegs.ax = 1                 'Visualizza puntatore mouse
          CALL INTERRUPT(&H33, InRegs, OutRegs)
       CASE 3
          Back% = Back% + 1
          IF Back% > 7 THEN Back% = 0
          COLOR Fore%, Back%
       CASE 4
          DrawFlag% = 1
       CASE 5
          LineFlag% = 1
```

continua

Listato 5.1 *Il programma Paint con gestione del mouse*

```
      CASE 6
         BoxFlag% = 1
      CASE 7
         CircleFlag% = 1
      CASE 8
         PaintFlag% = 1
      CASE 9
         InRegs.ax = 2                 'Nasconde puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         GET (0, 8)-(639, 199), Storage
         DEF SEG = VARSEG(Storage(1))
         BSAVE "PAINT.DAT", VARPTR(Storage(1)), 62532
         DEF SEG
         InRegs.ax = 1                 'Visualizza puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
      CASE 10
         InRegs.ax = 2                 'Nasconde puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         DEF SEG = VARSEG(Storage(1))
         BLOAD "PAINT.DAT", VARPTR(Storage(1))
         DEF SEG
         PUT (0, 8), Storage, PSET
         InRegs.ax = 1                 'Visualizza puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
      END SELECT
   RETURN

DrawPixel:
      InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
      InRegs.ax = 6
      CALL INTERRUPT(&H33, InRegs, OutRegs)

      DO
         InRegs.ax = 3
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         X% = OutRegs.cx
         Y% = OutRegs.dx
         InRegs.ax = 2                 'Nasconde puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         PSET (X%, Y%)
         InRegs.ax = 1                 'Visualizza puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         InRegs.bx = 0               'Rilascio pulsante sinistro
         InRegs.ax = 6
         CALL INTERRUPT(&H33, InRegs, OutRegs)
      LOOP WHILE OutRegs.bx = 0
```

continua

Listato 5.1 *Il programma Paint con gestione del mouse*

```
    RETURN

DrawLine:
      X% = OutRegs.cx
      Y% = OutRegs.dx
      XOld% = X%
      YOld% = Y%

      InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
      InRegs.ax = 6
      CALL INTERRUPT(&H33, InRegs, OutRegs)

      DO
         InRegs.ax = 3
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         YNew% = OutRegs.dx
         XNew% = OutRegs.cx
         InRegs.ax = 2                 'Nasconde puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         LINE (X%, Y%)-(XOld%, YOld%), Back%
         LINE (X%, Y%)-(XNew%, YNew%), Fore%
         XOld% = XNew%
         YOld% = YNew%
         InRegs.ax = 1                 'Visualizza puntatore mouse
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         InRegs.bx = 0              'Rilascio pulsante sinistro
         InRegs.ax = 6
         CALL INTERRUPT(&H33, InRegs, OutRegs)
      LOOP WHILE OutRegs.bx = 0
    RETURN

DrawBox:
      X% = OutRegs.cx
      Y% = OutRegs.dx
      XOld% = X%
      YOld% = Y%

      InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
      InRegs.ax = 6
      CALL INTERRUPT(&H33, InRegs, OutRegs)

      DO
         InRegs.ax = 3
         CALL INTERRUPT(&H33, InRegs, OutRegs)
         YNew% = OutRegs.dx
         XNew% = OutRegs.cx
```

continua

Listato 5.1 *Il programma Paint con gestione del mouse*

```
        InRegs.ax = 2                    'Nasconde puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        LINE (X%, Y%)-(XOld%, YOld%), Back%, B
        LINE (X%, Y%)-(XNew%, YNew%), Fore%, B
        XOld% = XNew%
        YOld% = YNew%
        InRegs.ax = 1                    'Visualizza puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        InRegs.bx = 0                    'Rilascio pulsante sinistro
        InRegs.ax = 6
        CALL INTERRUPT(&H33, InRegs, OutRegs)
      LOOP WHILE OutRegs.bx = 0
    RETURN

DrawCircle:
      X% = OutRegs.cx
      Y% = OutRegs.dx
      XOld% = X%
      YOld% = Y%

      InRegs.bx = 0  'Il rilascio del pulsante svuota la coda
      InRegs.ax = 6
      CALL INTERRUPT(&H33, InRegs, OutRegs)

      DO
        InRegs.ax = 3
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        YNew% = OutRegs.dx
        XNew% = OutRegs.cx
        InRegs.ax = 2                      'Nasconde puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        CIRCLE (X%, Y%), SQR((X% - XOld%) ^ 2 + (Y% - YOld%)]
                ^ 2),_
           Back%
        CIRCLE (X%, Y%), SQR((X% - XNew%) ^ 2 + (Y% - YNew%)]
                ^ 2)
        XOld% = XNew%
        YOld% = YNew%
        InRegs.ax = 1                    'Visualizza puntatore mouse
        CALL INTERRUPT(&H33, InRegs, OutRegs)
        InRegs.bx = 0                    'Rilascio pulsante sinistro
        InRegs.ax = 6
        CALL INTERRUPT(&H33, InRegs, OutRegs)
      LOOP WHILE OutRegs.bx = 0
    RETURN
```

continua

Listato 5.1 *Il programma Paint con gestione del mouse*

```
DrawPaint:
      X% = OutRegs.cx
      Y% = OutRegs.dx
      InRegs.ax = 2                    'Nasconde puntatore mouse
      CALL INTERRUPT(&H33, InRegs, OutRegs)
      PAINT (X%, Y%)
      InRegs.ax = 1                    'Visualizza puntatore mouse
      CALL INTERRUPT(&H33, InRegs, OutRegs)
   RETURN
```

Listato 5.1 *Il programma Paint con gestione del mouse.*

Si potrà ora avviare il programma e vedere come è possibile creare elementi grafici e applicare loro un determinato colore.
Si passerà, ora, a un'ulteriore trattazione della grafica che prevede l'animazione, ma, prima, si desidera presentare un'ulteriore istruzione (*DRAW*).

DISEGNO DI UN OROLOGIO

L'istruzione *DRAW* consente di produrre elementi grafici *vettoriali*. L'istruzione *DRAW* ha come argomento una *stringa*: per esempio, l'istruzione di seguito riportata genera la realizzazione a video di un esagono.

```
DRAW "BU50 NL25 F12 D20 G12 L50 H12 U20 E12 R25 BD22"
```

Consiglio: L'istruzione *DRAW* del BASIC è uno dei sistemi più compatti per produrre elementi grafici.

Ogni lettera corrisponde a un'impostazione di *DRAW* e ogni numero corrisponde alla lunghezza dell'elemento grafico tracciato (espresso in pixel). Di seguito viene indicato il significato di ciascuna lettera.

U	Su (*Up*)
D	Giù (*Down*)
L	Sinistra (*Left*)
R	Destra (*Right*)
E	In alto e a destra (*Up* e *Right*)
F	In basso e a destra (*Down* e *Right*)
G	In basso e a sinistra (*Down* e *Left*)
H	In alto e a sinistra (*Up* e *Left*)

B	Solleva la "penna" (non disegna)
N	Abbassa la "penna" (disegna)

La tabella sopra riportata consente quindi di poter decifrare l'istruzione *DRAW*. Quando questa istruzione viene utilizzata per la prima volta, la "penna" viene automaticamente posizionata al centro dello schermo. Da questa posizione essa verrà prelevata (mantenendola sollevata per non generare alcun elemento grafico) e verrà spostata di 50 pixel dove viene abbassata e spostata a sinistra di 25 pixel:

```
DRAW "BU50 NL25 ..."
```

Successivamente la penna viene spostata diagonalmente verso il basso e a destra di 12 pixel e in seguito di nuovo verticalmente verso il basso di 20 pixel:

```
DRAW "BU50 NL25 F12 D20 ..."
```

A questo punto la "penna" viene spostata diagonalmente verso il basso e a sinistra di 12 pixel, orizzontalmente a sinistra di 20 e così via fino a quando l'esagono non sarà completato:

```
DRAW "BU50 NL25 F12 D20 G12 L50 H12 U20 E12 R25 BD22"
```

In fase di creazione di un elemento grafico come quello sopra definito, si tenga presente che potranno essere realizzati angoli del valore pari a *0, 45* o *90* gradi. Si potrà, tuttavia, utilizzare l'argomento *"TA="* per *applicare una rotazione* alle azioni grafiche successivamente avviate. Per esempio, l'istruzione di seguito riportata traccia una linea verticale verso l'alto lunga 50 pixel: vale a dire con un angolo di 90 gradi.

```
DRAW "NU50"
```

Tuttavia, se alla stessa linea si desidera applicare un'angolo di 95 gradi, sarà necessario spostare gli assi di 5 gradi in senso antiorario:

```
ANG = 5
DRAW "TA="+VARPTR$(ANG)+" NU50"
```

Con questa particolare forma, che trasferisce un puntatore di stringa a una variabile che contiene la correzione dell'angolo, viene richiesta quando si desidera utilizzare l'argomento *TA=*. In questo caso si è corretto l'angolo verticale spostandolo in senso antiorario di 5 gradi perciò la linea verrà tracciata a 95 gradi. Un valore negativo correggerà l'angolo di tracciamento della linea in senso orario. Si tenga presente che la correzione dell'angolo di tracciamento rimarrà attiva fino a quando questa non verrà deliberatamente modificata.

Si provi ora a realizzare un ipotetico orologio che sia in grado di visualizzare l'ora corrente. La prima operazione da compiere è disegnare l'esagono e in seguito spostare la "penna" al centro dell'esagono.

```
SCREEN 8
DRAW "BU50 NL25 F12 D20 G12 L50 H12 U20 E12 R25 BD22"
```

È ora necessario calcolare l'angolo (rispetto alla verticale) sia della lancetta dei minuti che per quella delle ore utilizzando la funzione *TIMER* del BASIC (notare che vengono creati angoli con valore negativo poiché si desidera generare angolazioni in senso orario).

```
SCREEN 8
DRAW "BU50 NL25 F12 D20 G12 L50 H12 U20 E12 R25 BD22"
TimeMark! = TIMER
Hours! = INT(TimeMark! / 3600)
Remainder! = TimeMark - 3600 * Hours!
IF Hours! > 12 THEN Hours! = Hours! - 12
HourAngle! = -Hours! / 12 * 360
Minutes! = INT(Remainder! / 60)
MinuteAngle! = -Minutes! / 60 * 360
   :
   :
```

Infine, sarà sufficiente definire le lancette dei minuti e delle ore:

```
SCREEN 8
DRAW "BU50 NL25 F12 D20 G12 L50 H12 U20 E12 R25 BD22"
TimeMark! = TIMER
Hours! = INT(TimeMark! / 3600)
Remainder! = TimeMark - 3600 * Hours!
IF Hours! > 12 THEN Hours! = Hours! - 12
HourAngle! = -Hours! / 12 * 360
Minutes! = INT(Remainder! / 60)
MinuteAngle! = -Minutes! / 60 * 360
DRAW "TA=" + VARPTR$(HourAngle!) + " NU8"
DRAW "TA=" + VARPTR$(MinuteAngle!) + " NU12"
```

Se necessario, questo programma potrà essere adattato in modo che l'immagine venga regolarmente aggiornata a video e fintanto che essa risulterà a video, verrà riportata l'ora corrente.
Ora che si è visto in generale come operare con l'istruzione *DRAW*, si passerà alla realizzazione dell'animazione e alla creazione di *sprite*.

SPRITE E ANIMAZIONE

Per *sprite* si indica un piccolo elemento grafico che potrà essere immesso in una qualsiasi posizione dello schermo: per esempio, nel programma che verrà ora presentato, verrà realizzato un piccolo missile a cui viene associato l'effetto di esplosione prodotto dal reattore posto in coda allo stesso. Dopo aver realizzato e salvato l'immagine, questa potrà essere posizionata in un qualsiasi punto dello schermo per associare all'elemento grafico una sorta di animazione.

Proprio come si è fatto con il programma *paint* precedentemente presentato, si potranno progettare e utilizzare gli *sprite* immettendole in particolari subroutine che verranno richiamate all'interno del corpo principale del programma (*SPRITE.BAS*). Si inizi con la realizzazione dello *sprite*. Per rendere più semplice l'intero processo, verrà utilizzato un array di tipo *STRING* e, poiché i valori utilizzabili per il colore vanno da *0* a *15* verranno utilizzate le appropriate cifre esadecimali per poter generare lo *sprite* operando pixel per pixel. Per esempio, il programma *SPRITE.BAS* può essere iniziato con il disegno del missile:

```
DIM SpriteArray(1 TO 16) AS STRING

   SpriteArray(1)  =  "0000000010000000"
   SpriteArray(2)  =  "0000000010000000"
   SpriteArray(3)  =  "0000000111000000"
   SpriteArray(4)  =  "0000001131100000"
   SpriteArray(5)  =  "0000001131100000"
   SpriteArray(6)  =  "0000011333110000"
   SpriteArray(7)  =  "0000011333110000"
   SpriteArray(8)  =  "0010011333110010"
   SpriteArray(9)  =  "0011001131100110"
   SpriteArray(10) = "0011101131101110"
   SpriteArray(11) = "0011111131111110"
   SpriteArray(12) = "0011100131001110"
   SpriteArray(13) = "0011001111100110"
   SpriteArray(14) = "0010000222000010"
   SpriteArray(15) = "0000000222000000"
   SpriteArray(16) = "0000000020000000"
   :
   :
```

***Nota**: Se il campo dei valori per l'assegnazione dei colori viene ristretto all'intervallo che va da 0 a 15 (vale a dire da 0 a &HF), si limiterà l'effetto massimo ottenibile in modalità VGA (che prevede un intervallo che va da 0 a &HFF).*

Ora sarà necessario convertire il video in modalità grafica e utilizzare una subroutine che disegni sullo schermo il missile, convertendo ciascuna stringa in pixel:

```
DIM SpriteArray(1 TO 16) AS STRING

   SpriteArray(1)  =  "0000000010000000"
   SpriteArray(2)  =  "0000000010000000"
   SpriteArray(3)  =  "0000000111000000"
   SpriteArray(4)  =  "0000001131100000"
   SpriteArray(5)  =  "0000001131100000"
   SpriteArray(6)  =  "0000011333110000"
   SpriteArray(7)  =  "0000011333110000"
   SpriteArray(8)  =  "0010011333110010"
```

```
SpriteArray(9) =  "0011001131100110"
SpriteArray(10) = "0011101131101110"
SpriteArray(11) = "0011111131111110"
SpriteArray(12) = "0011100131001110"
SpriteArray(13) = "0011001111100110"
SpriteArray(14) = "0010000222000010"
SpriteArray(15) = "0000000222000000"
SpriteArray(16) = "0000000020000000"

 screenMode% = 1
GOSUB DrawSprite
:
:
```

Ora che il missile è stato riportato a video, esso potrà venire memorizzato in un array utilizzando una subroutine a cui si potrebbe assegnare il nome *GetSprite*. Per eseguire questa operazione sarà necessario prima dimensionare l'array in modo che questo possa contenere i dati relativi allo *sprite*. All'array si potrebbe assegnare il nome *GetArray( )*. Poiché si desidera modificare la dimensione del missile man mano che questo viene disegnato, l'array dovrà essere di tipo *dinamico*: questo significa che la sua dimensione verrà regolata dall'istruzione *REDIM* che deve essere immessa in *GetSprite*:

```
' $DYNAMIC

DIM SpriteArray(1 TO 16) AS STRING
DIM GetArray(1) AS INTEGER

SpriteArray(1) =  "0000000010000000"
SpriteArray(2) =  "0000000010000000"
SpriteArray(3) =  "0000000111000000"
SpriteArray(4) =  "0000001131100000"
SpriteArray(5) =  "0000001131100000"
SpriteArray(6) =  "0000011333110000"
SpriteArray(7) =  "0000011333110000"
SpriteArray(8) =  "0010011333110010"
SpriteArray(9) =  "0011001131100110"
SpriteArray(10) = "0011101131101110"
SpriteArray(11) = "0011111131111110"
SpriteArray(12) = "0011100131001110"
SpriteArray(13) = "0011001111100110"
SpriteArray(14) = "0010000222000010"
SpriteArray(15) = "0000000222000000"
SpriteArray(16) = "0000000020000000"
ScreenMode% = 1
GOSUB DrawSprite
GOSUB GetSprite
:
```

Consiglio: Il ridimensionamento dinamico degli array con *REDIM* è una nuova funzione del BASIC. *REDIM* deve essere utilizzato soprattutto quando non si possiede una notevole quantità di memoria — la dimensione di un array potrà essere incrementata o diminuita in base alla memoria disponibile. Il BASIC PDS va ancora oltre fornendo la funzione *SETMEM( )* che consente un maggiore controllo sulla memoria.

Come si è già fatto per il precedente programma *paint*, si dovrà ora memorizzare lo *sprite* su disco per poterlo in seguito richiamare. Sarà quindi necessario realizzare due ulteriori subroutine (*SaveSprite* e *LoadSprite*) che, rispettivamente, salveranno e richiameranno in memoria lo *sprite*.

```
$DYNAMIC

DIM SpriteArray(1 TO 16) AS STRING
DIM GetArray(1) AS INTEGER

  SpriteArray(1) =  "0000000010000000"
  SpriteArray(2) =  "0000000010000000"
  SpriteArray(3) =  "0000000111000000"
  SpriteArray(4) =  "0000001131100000"
  SpriteArray(5) =  "0000001131100000"
  SpriteArray(6) =  "0000011333110000"
  SpriteArray(7) =  "0000011333110000"
  SpriteArray(8) =  "0010011333110010"
  SpriteArray(9) =  "0011001131100110"
  SpriteArray(10) = "0011101131101110"
  SpriteArray(11) = "0011111131111110"
  SpriteArray(12) = "0011100131001110"
  SpriteArray(13) = "0011001111100110"
  SpriteArray(14) = "0010000222000010"
  SpriteArray(15) = "0000000222000000"
  SpriteArray(16) = "0000000020000000"

  ScreenMode% = 1
  GOSUB DrawSprite
  GOSUB GetSprite
  GOSUB SaveSprite
  GOSUB LoadSprite
  :
  :
```

Per essere sicuri che lo sprite venga nuovamente caricato in memoria intatto, esso potrà venire visualizzato sullo schermo con l'istruzione *PUT* del BASIC, che, come riportato di seguito, lo posizionerà a *(50, 50)*.

```
' $DYNAMIC

DIM SpriteArray(1 TO 16) AS STRING
DIM GetArray(1) AS INTEGER

   SpriteArray(1) =  "0000000010000000"
   SpriteArray(2) =  "0000000010000000"
   SpriteArray(3) =  "0000000111000000"
   SpriteArray(4) =  "0000001131100000"
   SpriteArray(5) =  "0000001131100000"
   SpriteArray(6) =  "0000011333110000"
   SpriteArray(7) =  "0000011333110000"
   SpriteArray(8) =  "0010011333110010"
   SpriteArray(9) =  "0011001131100110"
   SpriteArray(10) = "0011101131101110"
   SpriteArray(11) = "0011111131111110"
   SpriteArray(12) = "0011100131001110"
   SpriteArray(13) = "0011001111100110"
   SpriteArray(14) = "0010000222000010"
   SpriteArray(15) = "0000000222000000"
   SpriteArray(16) = "0000000020000000"

   ScreenMode% = 1
   GOSUB DrawSprite

   GOSUB GetSprite

   GOSUB SaveSprite

   GOSUB LoadSprite

   PUT (50, 50), GetArray
   :
   :
```

Ora si è pronti ad applicare l'animazione. È necessario creare un *loop* che richiami lo *sprite* a video, lo cancelli, lo sposti leggermente verso l'alto e ripeta continuamente questa procedura in modo che si abbia l'impressione che il missile si stia spostando verso l'alto. Come precedentemente accennato, il metodo di gestione di *PUT* prevede un'operazione di tipo *XOR* sui bit di un'immagine a video quando ve ne è una precedente già visualizzata. Questo significa che l'istruzione *PUT* viene utilizzata sempre per la stessa immagine che verrà visualizzata nella sua posizione precedente anche in un secondo momento, verrà successivamente cancellata la precedente posizione (qualsiasi numero che viene soggetto a un'operazione di tipo *XOR* su se stesso è pari a *0*). A questo punto il programma sarà in grado di ridisegnare il missile in una posizione leggermente superiore rispetto alla precedente, in seguito cancellandola, e così via. Al termine, sarà necessario disegnare il missile per un'ultima volta

(poiché ogni precedente immagine viene cancellata). Il listato 5.2 presenta l'intero programma.

```
' $DYNAMIC

DIM SpriteArray(1 TO 16) AS STRING
DIM GetArray(1) AS INTEGER

    SpriteArray(1) =  "0000000010000000"
    SpriteArray(2) =  "0000000010000000"
    SpriteArray(3) =  "0000000111000000"
    SpriteArray(4) =  "0000001131100000"
    SpriteArray(5) =  "0000001131100000"
    SpriteArray(6) =  "0000011333110000"
    SpriteArray(7) =  "0000011333110000"
    SpriteArray(8) =  "0010011333110010"
    SpriteArray(9) =  "0011001131100110"
    SpriteArray(10) = "0011101131101110"
    SpriteArray(11) = "0011111131111110"
    SpriteArray(12) = "0011100131001110"
    SpriteArray(13) = "0011001111100110"
    SpriteArray(14) = "0010000222000010"
    SpriteArray(15) = "0000000222000000"
    SpriteArray(16) = "0000000020000000"

    ScreenMode% = 1
    GOSUB DrawSprite
    GOSUB GetSprite
    GOSUB SaveSprite
    GOSUB LoadSprite

    PUT (50, 50), GetArray

    FOR i = 1 TO 100
       PUT (100, 100 - i), GetArray
       PUT (100, 100 - i), GetArray
    NEXT i

    PUT (100, 0), GetArray

    END
```

Listato 5.2 *Il programma SPRITE.BAS per disegnare e applicare l'animazione agli sprite.*

Completato il programma *SPRITE.BAS*, che comprende l'animazione, non rimane ora che realizzare la subroutine.

LA SUBROUTINE *DrawSprite*

Poiché si suppone di dover decodificare lo sprite definito nell'array *SpriteArray( )* (di tipo *STRING*) sarà necessario iniziare impostando lo schermo in modo che questo operi nella modalità grafica che è stata definita da *ScreenMode%* e che venga controllata la posizione dove verrà visualizzato lo sprite: per esempio, *(10, 10)*.

```
DrawSprite:
   SCREEN ScreenMode%
   XStart = 10
   YStart = 10
   :
```

In seguito, è necessario decifrare le stringhe dell'array *SpriteArray( )* e utilizzare l'istruzione *PSET* del BASIC per attivarne i relativi pixel. Il valore del colore del pixel verrà decodificato avviando un *loop* su ogni stringa dell'array — e mantenendo attivo il *loop* su ogni carattere della stringa — in modo da poter impostare il valore del colore del pixel attivo e trasferirlo alla variabile *CurrentColor%*.

```
DrawSprite:
   SCREEN ScreenMode%
   XStart = 10
   YStart = 10
   FOR i = 1 TO UBOUND(SpriteArray, 1)
      FOR j = 1 TO LEN(SpriteArray(i))
         CurrentColor% = ASC(UCASE$(MID$(SpriteArray(i), j, 1)))
         :
         :
      NEXT j
   NEXT i
   :
   :
```

Ora che il carattere (*in notazione esadecimale*) è stato trasferito a *CurrentColor%*, sarà necessario decodificarlo in un valore di colore che vada da *0* a *15*. Si dovrà, in questo caso, utilizzare *PSET( )* per poter attivare l'appropriato pixel:

```
DrawSprite:
   SCREEN ScreenMode%
   XStart = 10
   YStart = 10

   FOR i = 1 TO UBOUND(SpriteArray, 1)
      FOR j = 1 TO LEN(SpriteArray(i))
         CurrentColor% = ASC(UCASE$(MID$(SpriteArray(i), j, 1)))
         IF CurrentColor% <= ASC("9") THEN
            CurrentColor% = CurrentColor% - ASC("0")
         ELSE
            CurrentColor% = CurrentColor% - ASC("A") + 10
```

```
            END IF
            PSET (XStart + j - 1, YStart + i - 1), CurrentColor%
         NEXT j
      NEXT i
      :
      :
```

A questo punto il missile dovrebbe essere visibile a video: il listato 5.3 mostra l'intera subroutine.

```
DrawSprite:
    SCREEN ScreenMode%
    XStart = 10
    YStart = 10
    FOR i = 1 TO UBOUND(SpriteArray, 1)
       FOR j = 1 TO LEN(SpriteArray(i))
          CurrentColor% = ASC(UCASE$(MID$(SpriteArray(i), j, 1)))
          IF CurrentColor% <= ASC("9") THEN
             CurrentColor% = CurrentColor% - ASC("0")
          ELSE
             CurrentColor% = CurrentColor% - ASC("A") + 10
          END IF
          PSET (XStart + j - 1, YStart + i - 1), CurrentColor%
       NEXT j
    NEXT i
    RETURN
```

Listato 5.3 *La subroutine DrawSprite*

LA SUBROUTINE *GetSprite*

Lo scopo di questa subroutine è quello di leggere lo *sprite* presente a video e memorizzarne i dati nell'array *GetArray( )*. Vi sono formule standard per determinare la quantità di memoria necessaria per memorizzare uno sprite di una determinata dimensione, e queste sono in stretta relazione con la modalità video.

Quello che è ora necessario è conoscere la dimensione dello *sprite* espressa in pixel, che potrà essere ottenuta dall'associazione delle dimensioni dell'array *SpriteArray( )* e della modalità video, che dovrebbe ancora essere contenuta nella variabile *Screen Mode%*.

Sarà necessario inoltre determinare il numero di word a 16 bit che saranno necessarie per memorizzare lo *sprite* e posizionarne il valore in *SpriteSize*.

```
GetSprite:
   Cols = LEN(SpriteArray(1))
   Rows = UBOUND (SpriteArray, 1)

   SELECT CASE ScreenMode%
      CASE 1
         SpriteSize = (4+INT(((Rows+1)*2+7)/8)*1*(Cols+1) + 1) \ 2
      CASE 2, 11
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*1*(Cols+1) + 1) \ 2
      CASE 7, 8, 9, 12
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*4*(Cols+1) + 1) \ 2
      CASE 10
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*2*(Cols+1) + 1) \ 2
      CASE 13
         SpriteSize = (4+INT(((Rows+1)*8+7)/8)*1*(Cols+1) + 1) \ 2
      CASE ELSE
         SpriteSize = 0
   END SELECT
   :
   :
```

Ora sarà possibile ridimensionare *GetArray( )* poiché è noto quante word da 16 bit — vale a dire valori di tipo *INTEGER* — sono necessarie. Si potrà immediatamente usare l'istruzione *REDIM* per ridimensionare *GetArray( )* (si noti che si dovrà tuttavia utilizzare l'istruzione *' $DYNAMIC* all'inizio del programma SPRITE.BAS in modo da rendere *dinamico* l'array).

```
GetSprite:
   Cols = LEN(SpriteArray(1))
   Rows = UBOUND (SpriteArray, 1)

   SELECT CASE ScreenMode%
      CASE 1
         SpriteSize = (4+INT(((Rows+1)*2+7)/8)*1*(Cols+1) + 1) \ 2
      CASE 2, 11
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*1*(Cols+1) + 1) \ 2
      CASE 7, 8, 9, 12
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*4*(Cols+1) + 1) \ 2
      CASE 10
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*2*(Cols+1) + 1) \ 2
      CASE 13
         SpriteSize = (4+INT(((Rows+1)*8+7)/8)*1*(Cols+1) + 1) \ 2
      CASE ELSE
         SpriteSize = 0
   END SELECT
   REDIM GetArray(1 TO SpriteSize) AS INTEGER
   :
   :
```

Per memorizzare lo sprite, sarà ora sufficiente utilizzare l'istruzione *GET*. Il listato 5.4 presenta la subroutine *GetSprite* completa.

```
GetSprite:
    Cols = LEN(SpriteArray(1))
    Rows = UBOUND (SpriteArray, 1)

    SELECT CASE ScreenMode%
       CASE 1
         SpriteSize = (4+INT(((Rows+1)*2+7)/8)*1*(Cols+1) + 1) \ 2
       CASE 2, 11
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*1*(Cols+1) + 1) \ 2
       CASE 7, 8, 9, 12
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*4*(Cols+1) + 1) \ 2
       CASE 10
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*2*(Cols+1) + 1) \ 2
       CASE 13
         SpriteSize = (4+INT(((Rows+1)*8+7)/8)*1*(Cols+1) + 1) \ 2
       CASE ELSE
         SpriteSize = 0
    END SELECT
    REDIM GetArray(1 TO SpriteSize) AS INTEGER
    GET (10, 10)-(10 + Rows, 10 + Cols), GetArray
    RETURN
```

Listato 5.4 *La subroutine GetSprite.*

Ora lo sprite è stato memorizzato in *GetArray( )*. Le due successive subroutine consentiranno di memorizzare su disco lo *sprite* e, successivamente di ricaricarlo in memoria.

LA SUBROUTINE *SaveSprite*

Per salvare lo *sprite* su disco, sarà sufficiente utilizzare l'istruzione *BSAVE*, come riportato nel listato 5.5.

```
SaveSprite:
    DEF SEG = VARSEG(GetArray(1))
    BSAVE "SPRITE.DAT", VARPTR(GetArray(1)), 2 * UBOUND
                        (GetArray, 1)
    DEF SEG
    RETURN
```

Listato 5.5 *La subroutine SaveSprite.*

LA SUBROUTINE *LoadSprite*

La subroutine finale dello *sprite* prevede il suo nuovo invio *GetArray( )* tramite l'istruzione *BLOAD*, che è già stata trattata nel precedente programma *paint*. Il processo di caricamento viene riportato nel listato 5.6.

```
LoadSprite:

    DEF SEG = VARSEG(GetArray(1))
    BLOAD "SPRITE.DAT", VARPTR(GetArray(1))
    DEF SEG
    RETURN
```

Listato 5.6 *La subroutine LoadSprite.*

IL LISTATO COMPLETO DEL PROGRAMMA SPRITE.BAS

Il listato 5.7 mostra l'intero programma *SPRITE.BAS* completo delle procedure di animazione, riportate alla fine del corpo principale del programma stesso.

```
' $DYNAMIC
DIM SpriteArray(1 TO 16) AS STRING
DIM GetArray(1) AS INTEGER

    SpriteArray(1) =  "0000000010000000"
    SpriteArray(2) =  "0000000010000000"
    SpriteArray(3) =  "0000000111000000"
    SpriteArray(4) =  "0000001131100000"
    SpriteArray(5) =  "0000001131100000"
    SpriteArray(6) =  "0000011333110000"
    SpriteArray(7) =  "0000011333110000"
    SpriteArray(8) =  "0010011333110010"
    SpriteArray(9) =  "0011001131100110"
    SpriteArray(10) = "0011101131101110"
    SpriteArray(11) = "0011111131111110"
    SpriteArray(12) = "0011100131001110"
    SpriteArray(13) = "0011001111100110"
    SpriteArray(14) = "0010000222000010"
    SpriteArray(15) = "0000000222000000"
    SpriteArray(16) = "0000000020000000"
```

continua

Listato 5.7 *Il listato completo del programma SPRITE.BAS*

```
    ScreenMode% = 1

    GOSUB DrawSprite
    GOSUB GetSprite
    GOSUB SaveSprite
    GOSUB LoadSprite

    PUT (50, 50), GetArray

    FOR i = 1 TO 100
       PUT (100, 100 - i), GetArray
       PUT (100, 100 - i), GetArray
    NEXT i

    PUT (100, 0), GetArray

    END

DrawSprite:
    SCREEN ScreenMode%
    XStart = 10
    YStart = 10
    FOR i = 1 TO UBOUND(SpriteArray, 1)
       FOR j = 1 TO LEN(SpriteArray(i))
          CurrentColor% = ASC(UCASE$(MID$(SpriteArray(i), j, 1)))
          IF CurrentColor% <= ASC("9") THEN
             CurrentColor% = CurrentColor% - ASC("0")
          ELSE
             CurrentColor% = CurrentColor% - ASC("A") + 10
          END IF
          PSET (XStart + j - 1, YStart + i - 1), CurrentColor%
       NEXT j
    NEXT i
    RETURN

GetSprite:
    Cols = LEN(SpriteArray(1))
    Rows = UBOUND (SpriteArray, 1)

    SELECT CASE ScreenMode%
       CASE 1
         SpriteSize = (4+INT(((Rows+1)*2+7)/8)*1*(Cols+1) + 1) \ 2
       CASE 2, 11
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*1*(Cols+1) + 1) \ 2
       CASE 7, 8, 9, 12
         SpriteSize = (4+INT(((Rows+1)*1+7)/8)*4*(Cols+1) + 1) \ 2
```

continua

Listato 5.7 *Il listato completo del programma SPRITE.BAS*

```
        CASE 10
          SpriteSize = (4+INT(((Rows+1)*1+7)/8)*2*(Cols+1) + 1) \ 2
        CASE 13
          SpriteSize = (4+INT(((Rows+1)*8+7)/8)*1*(Cols+1) + 1) \ 2
        CASE ELSE
          SpriteSize = 0
    END SELECT
    REDIM GetArray(1 TO SpriteSize) AS INTEGER
    GET (10, 10)-(10 + Rows, 10 + Cols), GetArray
    RETURN

SaveSprite:
    DEF SEG = VARSEG(GetArray(1))
    BSAVE "SPRITE.DAT", VARPTR(GetArray(1)), 2 * UBOUND
                                (GetArray, 1)
    DEF SEG
    RETURN

LoadSprite:

    DEF SEG = VARSEG(GetArray(1))
    BLOAD "SPRITE.DAT", VARPTR(GetArray(1))
    DEF SEG
    RETURN
```

Listato 5.7 *Il listato completo del programma SPRITE.BAS.*

In questo programma viene prima progettato e disegnato lo *sprite* e in seguito esso viene caricato in *GetArray( )*. La subroutine *SaveSprite* consente il salvataggio dei dati su disco e *LoadSprite* ne consente il nuovo caricamento in memoria. Infine, si è visto come è possibile animare il missile avviando un *loop* sulle istruzioni *PUT( )*. Si provi ad avviare il programma e osservare come questo genera un missile che dà l'impressione di essere lanciato verso l'alto.
Il prossimo paragrafo tratterà dell'uso del BASIC PDS per la gestione della grafica.

PRESENTATION GRAPHICS DEL BASIC PDS

Nel caso in cui si sia in possesso del BASIC PDS, vi è un'altro tipo di rappresentazione grafica alla quale è necessario accennare, specialmente se si stanno creando programmi di tipo finanziario: *Presentation Graphics*. Questa è una serie di routine che sono in grado di fornire i dati utilizzando vari tipi di rappresentazioni grafiche di tipo

finanziario. Per esempio, si presupponga di dover analizzare il consumo del burro, suddiviso per regione geografica in base ai dati di seguito riportati.

Regione	*Consumo burro (ton.)*
Nord	219
Est	19
Sud	119
Ovest	319

L'uso degli strumenti offerti da *Presentation Graphics* del BASIC PDS consente di visualizzare graficamente i dati sopra esposti sotto forma di grafico a torta, a barre o a linee. In questa parte del capitolo verranno prese in esame tutti e tre i tipi di rappresentazione grafica dei dati.

GRAFICI A TORTA

Per iniziare, è necessario osservare che se si sta utilizzando QuickBasic *QBX.EXE* si dovrà caricare anche *CHRTBEFR.QLB* che contiene una speciale libreria *Presentation Graphics* (si supponga di dover caricare un programma a cui è stato assegnato il nome *PIE.BAS*).

```
QBX PIE /L CHRTBEFR
```

D'altra parte, se invece si sta utilizzando *BC.EXE* (versione 7.0 o successiva) sarà necessario prima avviare un'operazione di *link* con la libreria *CHRTBEFR.LIB*:

```
LINK PIE,,,CHRTBEFR;
```

Si vedrà ora come sviluppare un programma che generi un grafico a torta (*PIE.BAS*): è necessario, innanzitutto, includere due file *.BI* che vengono forniti con il BASIC PDS:

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRB.BI'
   :
   :
```

In seguito, sarà necessario caricare il nome di ciascuna delle quattro regioni geografiche all'interno di un array (a cui viene assegnato il nome *Regions( )*), il consumo del burro per ogni regione geografica in un secondo array (*Consumption( )*) e, infine, creare un ultimo array a cui viene associato il nome *Exploded( )*.

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'

   DIM Regions(4) AS STRING
   DIM Consumption(4) AS SINGLE
```

```
DIM Exploded(4) AS INTEGER
:
:
```

Questo array finale è necessario unicamente per i *grafici a torta*. Se il valore in *Exploded( )* corrisponde a un particolare settore del grafico diverso da zero, tale settore verrà generato come "esploso", ovvero viene disegnato staccato rispetto al resto del grafico a torta. Per ora si lasceranno i valori di *Exploded( )* a *0*, il che significa che nessun settore della torta verrà esploso. Riempire quindi gli array come di seguito riportato:

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'

  DIM Regions(4) AS STRING
  DIM Consumption(4) AS SINGLE
  DIM Exploded(4) AS INTEGER

  Regions(1) = "Nord"
  Regions(2) = "Est"
  Regions(3) = "Sud"
  Regions(4) = "Ovest"

  Consumption(1) = 219
  Consumption(2) = 19
  Consumption(3) = 119
  Consumption(4) = 319
  :
  :
```

Si deve ora selezionare una modalità grafica, e per questa operazione non è sufficiente utilizzare *SCREEN( )*, ma sarà necessario utilizzare la nuova istruzione di BASIC PDS *ChartScreen( )*.
L'argomento che viene trasferito a *ChartScreen( )* deve corrispondere a una modalità video riconosciuta dal BASIC: nell'esempio qui riportato si presume di utilizzare la modalità *VGA* che corrisponde a *12*:

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'

  DIM Regions(4) AS STRING
  DIM Consumption(4) AS SINGLE
  DIM Exploded(4) AS INTEGER

  Regions(1) = "Nord"
  Regions(2) = "Est"
  Regions(3) = "Sud"
  Regions(4) = "Ovest"
```

```
Consumption(1) = 219
Consumption(2) = 19
Consumption(3) = 119
Consumption(4) = 319

CALL ChartScreen(12)
:
:
```

Ora si è pronti a specificare che tipo di grafico si dovrà generare. Questo potrà essere definito impostando una variabile di tipo *ChartEnvironment* che, in CHRTB.BI viene definita come di seguito riportato:

```
TYPE ChartEnvironment
   ChartType     AS INTEGER       ' 1=Bar, 2=Column, 3=Line,
                                    4=Scatter, 5=Pie
   ChartStyle    AS INTEGER       ' Dipende dal tipo
   DataFont      AS INTEGER       ' Font utilizzato per i caratteri
   ChartWindow   AS RegionType    ' Finestra generale del grafico
   DataWindow    AS RegionType    ' Porzione dei dati del grafico
   MainTitle    AS TitleType     ' Opzioni titolo principale
   SubTitle      AS TitleType     ' Opzioni seconda riga del titolo
   XAxis         AS AxisType      ' Opzioni asse delle X
   YAxis         AS AxisType      ' Opzioni asse delle Y
   Legend        AS LegendType    ' Opzioni delle legende
END TYPE
```

Una variabile di questo tipo dovrà essere trasferita agli strumenti grafici del BASIC PDS e l'impostazione dei campi in questa struttura di dati consente di specificare i titoli che dovranno essere immessi all'interno del grafico. Alla variabile *ChartEnvironment* potrà essere assegnato il nuovo nome *OurChart*.

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'

   DIM OurChart AS ChartEnvironment
   DIM Regions(4) AS STRING
   DIM Consumption(4) AS SINGLE
   DIM Exploded(4) AS INTEGER

   Regions(1) = "Nord"
   Regions(2) = "Est"
   Regions(3) = "Sud"
   Regions(4) = "Ovest"

   Consumption(1) = 219
   Consumption(2) = 19
   Consumption(3) = 119
   Consumption(4) = 319
```

```
CALL ChartScreen(12)
:
:
```

Sarà ora necessario indicare il *tipo* e lo *stile* di grafico che si intende ottenere e, a questo scopo, potrà essere utilizzata una delle seguenti costanti predefinite: *cBar, cColumn, cLine, cScatter* oppure *cPie*. Per l'esempio qui proposto sarà necessario utilizzare *cPie*. Sarà inoltre necessario selezionare lo stile del grafico selezionandolo una delle opzioni di seguito riportate:

Tipo	*Stile*
Barre	*cPlain* oppure *cStacked*
Colonne	*cPlain* oppure *cStacked*
Linea	*cLines* oppure *cNoLines*
Dispersione	*cLines* oppure *cNoLines*
Torta	*cPercent* oppure *cNoPercent*

Nell'esempio qui sviluppato verrà selezionato un grafico a torta (*cPie*) realizzato in uno stile che contrassegni ogni settore con un valore che sia una percentuale dell'intero (*cPercent*). Viene utilizzato a questo scopo il sottoprogramma del PDS *DefaultChart( )* per consentire a *Presentation Graphics* di riconoscere quali selezioni sono state effettuate:

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'

  DIM OurChart AS ChartEnvironment
  DIM Regions(4) AS STRING
  DIM Consumption(4) AS SINGLE
  DIM Exploded(4) AS INTEGER

  Regions(1) = "Nord"
  Regions(2) = "Est"
  Regions(3) = "Sud"
  Regions(4) = "Ovest"

  Consumption(1) = 219
  Consumption(2) = 19
  Consumption(3) = 119
  Consumption(4) = 319

  CALL ChartScreen(12)

  CALL DefaultChart(OurChart, cPie, cPercent)
  :
  :
```

Infine, dovranno essere completati i campi dei titoli della struttura dei dati *OurChart* per ottenere, finalmente la rappresentazione a video del grafico:

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'

   DIM OurChart AS ChartEnvironment
   DIM Regions(4) AS STRING
   DIM Consumption(4) AS SINGLE
   DIM Exploded(4) AS INTEGER

   Regions(1) = "Nord"
   Regions(2) = "Est"
   Regions(3) = "Sud"
   Regions(4) = "Ovest"

   Consumption(1) = 219
   Consumption(2) = 19
   Consumption(3) = 119
   Consumption(4) = 319

   CALL ChartScreen(12)

   CALL DefaultChart(OurChart, cPie, cPercent)

   OurChart.MainTitle.Title = "Consumo del burro"
   OurChart.MainTitle.TitleColor = 15
   OurChart.MainTitle.Justify = cCenter
   OurChart.SubTitle.Title = "Consumo (Ton.)"
   OurChart.SubTitle.TitleColor = 11
   OurChart.SubTitle.Justify = cCenter
   OurChart.ChartWindow.Border = cYes
   :
   :
```

Per visualizzare il grafico, a questo punto dovrà essere richiamato *ChartPie( )* con gli argomenti come di seguito specificato:

```
CALL ChartPie (OurChart, Regions( ), Consumption( ), Exploded( ), 4)
```

In questo caso, *OurChart* corrisponde alla variabile *ChartEnvironment* impostata, *Regions( )* contiene i titoli di ogni settore della torta, *Consumption( )* contiene i dati numerici da rappresentare graficamente, *Exploded( )* indica quale/i settore/i dovrà essere esploso e il valore intero finale, *4*, indica quante serie di dati vengono analizzate. Si può a questo punto richiamare *ChartPie( )* e in seguito concludere il programma come mostrato nel listato 5.8.

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'

    DIM OurChart AS ChartEnvironment
    DIM Regions(4) AS STRING
    DIM Consumption(4) AS SINGLE
    DIM Exploded(4) AS INTEGER

    Regions(1) = "Nord"
    Regions(2) = "Est"
    Regions(3) = "Sud"
    Regions(4) = "Ovest"

    Consumption(1) = 219
    Consumption(2) = 19
    Consumption(3) = 119
    Consumption(4) = 319

    CALL ChartScreen(12)

    CALL DefaultChart(OurChart, cPie, cPercent)

    OurChart.MainTitle.Title = "Consumo del burro"
    OurChart.MainTitle.TitleColor = 15
    OurChart.MainTitle.Justify = cCenter
    OurChart.SubTitle.Title = "Consumo (Ton.)"
    OurChart.SubTitle.TitleColor = 11
    OurChart.SubTitle.Justify = cCenter
    OurChart.ChartWindow.Border = cYes

    CALL ChartPie(OurChart, Regions(), Consumption(),
    Exploded(), 4)
    SLEEP
```

Listato 5.8 *Un programma PDS per generare un grafico a torta.*

Si noti che alla fine del programma è stata immessa l'istruzione *SLEEP* del BASIC che manterrà a video il grafico fintanto che non verrà premuto un tasto qualsiasi.

Consiglio: L'istruzione *SLEEP* del BASIC sostituisce la precedente istruzione *DO: LOOP WHILE INKEY$ = " "*. Se questa istruzione viene associata a un valore numerico, come, per esempio, *SLEEP 5*, il BASIC manterrà il grafico a video per il numero di secondi indicato (5).

A questo punto non si è fatto altro che creare la spina dorsale di un programma per la rappresentazione grafica dei dati. Il prossimo paragrafo prenderà in esame i grafici a barre.

I GRAFICI A BARRE

La creazione si un *grafico a barre* potrà prendere tranquillamente le mosse dal programma appena terminato. Sarà qui sufficiente eliminare la definizione di *Exploded( )* (che non ha nessun senso in un grafico di questo tipo) e specificare uno stile di grafico a barre in *DefaultChart( )*.

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'
  DIM OurChart AS ChartEnvironment
  DIM Regions(4) AS STRING
  DIM Consumption(4) AS SINGLE

  Regions(1) = "Nord"
  Regions(2) = "Est"
  Regions(3) = "Sud"
  Regions(4) = "Ovest"

  Consumption(1) = 219
  Consumption(2) = 19
  Consumption(3) = 119
  Consumption(4) = 319

  CALL ChartScreen(12)
  CALL DefaultChart(OurChart, cBar, cPlain)
  :
  :
```

L'operazione successiva, sarà quella di immettere i titoli per l'asse delle X e per l'asse delle Y (che, come è naturale, non avrebbero alcun senso in una rappresentazione grafica dei dati sotto forma di torta) e richiamare *Chart( )* — e non *ChartPie( )* — per visualizzare il grafico a barre. Il listato 5.9 riporta l'intero programma.

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'
  DIM OurChart AS ChartEnvironment
  DIM Regions(4) AS STRING
  DIM Consumption(4) AS SINGLE
```

continua

Listato 5.9 *Il programma per la generazione di un grafico a barre con il BASIC PDS.*

```
Regions(1) = "Nord"
Regions(2) = "Est"
Regions(3) = "Sud"
Regions(4) = "Ovest"

Consumption(1) = 219
Consumption(2) = 19
Consumption(3) = 119
Consumption(4) = 319

CALL ChartScreen(12)
CALL DefaultChart(OurChart, cBar, cPlain)

OurChart.MainTitle.Title = "Consumo del burro"
OurChart.MainTitle.TitleColor = 15
OurChart.MainTitle.Justify = cCenter
OurChart.SubTitle.Title = "Consumo (Ton.)"
OurChart.SubTitle.TitleColor = 11
OurChart.SubTitle.Justify = cCenter
OurChart.ChartWindow.Border = cYes

OurChart.XAxis.AxisTitle.Title = "Consumo"
OurChart.YAxis.AxisTitle.Title = "Regione geografica"

CALL Chart(OurChart, Regions(), Consumption(), Exploded(), 4)
SLEEP
```

Listato 5.9 *Il programma per la generazione di un grafico a barre con il BASIC PDS.*

I GRAFICI A LINEE

L'esempio sopra riportato potrà essere facilmente adattato per la generazione di un grafico a linee dove visualizzare tutti i punti collegati da una linea. Sarà sufficiente dichiarare questo tipo di grafico in *DefaultChart( )*.

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'

DIM OurChart AS ChartEnvironment
DIM Regions(4) AS STRING
DIM Consumption(4) AS SINGLE

Regions(1) = "Nord"
Regions(2) = "Est"
Regions(3) = "Sud"
Regions(4) = "Ovest"
```

```
Consumption(1) = 219
Consumption(2) = 19
Consumption(3) = 119
Consumption(4) = 319

CALL ChartScreen(12)
CALL DefaultChart(OurChart, cLine, cLine)
:
:
```

Sarà ora necessario di nuovo richiamare *Chart( )* (che traccia grafici a barre, a colonne o a linee in base a quanto richiesto in *DefaultChart( )*). Il listato 5.10 riporta il programma intero.

```
' $INCLUDE: 'C:\BC7\FONTB.BI'
' $INCLUDE: 'C:\BC7\CHRTB.BI'
   DIM OurChart AS ChartEnvironment
   DIM Regions(4) AS STRING
   DIM Consumption(4) AS SINGLE

   Regions(1) = "Nord"
   Regions(2) = "Est"
   Regions(3) = "Sud"
   Regions(4) = "Ovest"

   Consumption(1) = 219
   Consumption(2) = 19
   Consumption(3) = 119
   Consumption(4) = 319

   CALL ChartScreen(12)
   CALL DefaultChart(OurChart, cLine, cLine)

   OurChart.MainTitle.Title = "Consumo del burro"
   OurChart.MainTitle.TitleColor = 15
   OurChart.MainTitle.Justify = cCenter
   OurChart.SubTitle.Title = "Consumo (Ton.)"
   OurChart.SubTitle.TitleColor = 11
   OurChart.SubTitle.Justify = cCenter
   OurChart.ChartWindow.Border = cYes

   OurChart.XAxis.AxisTitle.Title = "Consumo"
   OurChart.YAxis.AxisTitle.Title = "Regione geografica"
   CALL Chart(OurChart, Regions(), Consumption(), Exploded(), 4)
   SLEEP
```

Listato 5.10 *Programma per generare con PDS una rappresentazione di dati sotto forma di grafico a linee.*

Indubbiamente, come si è potuto notare, il BASIC PDS fornisce validi strumenti per la rappresentazione grafica di serie di dati.
Prima di concludere l'analisi delle possibilità grafiche offerte dal BASIC, tuttavia, è qui necessario prendere in esame due ulteriori funzioni che consentono una gestione più corretta dell'ambiente grafico. La prima indicherà quale scheda grafica è stata installata nel computer e la seconda verificherà e comunicherà la risoluzione grafica (espressa in pixel) dello schermo.

CONTROLLO DELLA SCHEDA GRAFICA INSTALLATA

Si reputa estremamente utile concludere il capitolo che tratta della grafica con il BASIC con una funzione che consente di verificare quale scheda video è stata installata nel computer. In effetti, questa è un'operazione che è sempre stata difficile da eseguire e dopo aver dato un'occhiata al listato completo che verrà sviluppato, si capirà il perché. Tuttavia, la conoscenza della scheda video disponibile, indica quale modalità video si potrà utilizzare (è questa un'informazione indispensabile quando si intendono utilizzare le funzioni grafiche).

***Nota**: Con questa funzione non sarà più necessario richiedere all'utente di indicare quale scheda grafica è stata installata.*

Alla funzione viene assegnato il nome *GetVideoCard$( )*. Perché questa funzione abbia una sua utilità, sarà necessario che essa restituisca i seguenti valori in formato *STRING*:

```
GetVideoCard$      "MDA" = Monochrome display adapter
                   "CGA" = Color graphics adapter
                   "EGA" = Enhanced graphics adapter
                   "PGA" = Professional graphics adapter
                   "VGA" = Variable graphics array
```

***Nota**: La MDA è una speciale scheda, monocromatica, che è in grado di visualizzare la grafica solo in bianco e nero. Per questa ragione la MDA è in grado di supportare unicamente la modalità BIOS 7.*

La realizzazione di questa funzione richiede un po' di tempo. Innanzitutto sarà necessario verificare se il BIOS di sistema sia in grado di gestire una scheda VGA, per passare poi a una EGA, per scendere successivamente alla gestione delle schede CGA e MDA. Man mano che veniva prodotto un nuovo tipo di monitor, il BIOS è

stato portato a gestire un livello sempre superiore perciò, si dovrà interrogare direttamente il BIOS per controllare con quale versione si sta operando.
Innanzitutto, sarà necessario verificare che il BIOS all'interno del computer sia abbastanza avanzato da poter gestire l'interrupt &H10, servizio &H1A che è in grado di indicare esattamente quale tipo di scheda video è stata installata. Nel caso in cui si possa ottenere questa informazione, &H1A viene restituito in *al* e il codice video viene trasferito in *bl*. Per poter controllare questo tipo di codice si dovrà utilizzare un'istruzione di tipo *SELECT CASE*.

```
FUNCTION GetVideoCard$

   DIM InRegs AS RegType, OutRegs AS RegType
   InRegs.ax = &H1A00
   CALL INTERRUPT(&H10, InRegs, OutRegs)
   IF (OutRegs.ax AND &HFF) = &H1A THEN
      Code = OutRegs.bx AND &HFF
      SELECT CASE Code
         CASE 1
            GetVideoCard$ = "MDA"
         CASE 2
            GetVideoCard$ = "CGA"
         CASE 4 TO 5
            GetVideoCard$ = "EGA"
         CASE 6
            GetVideoCard$ = "PGA"
         CASE 7 TO 8
            GetVideoCard$ = "VGA"
         END SELECT
      EXIT FUNCTION
      :
      :
```

Tuttavia, solo i BIOS di realizzazione più recente (quelli in sostanza in grado di gestire una scheda video VGA) sono in grado di gestire questo servizio. Se questo servizio non può essere gestito, sarà necessario continuare la verifica passando alla scheda video EGA tramite l'interrupt &H10, servizio &H12. Se questo servizio viene invocato con *bl = &H10* e, il valore restituito in bl non è maggiore o uguale a &H10, questo sta a significare che è stata installata una scheda video EGA. Si noti che, a differenza del servizio &H1A, questo servizio indica solo la presenza o l'assenza di un singolo tipo di scheda video, la *EGA*. A questo punto, se viene riconosciuta la presenza di una scheda EGA si potrà uscire dalla funzione.

```
FUNCTION GetVideoCard$

   DIM InRegs AS RegType, OutRegs AS RegType
   InRegs.ax = &H1A00
   CALL INTERRUPT(&H10, InRegs, OutRegs)
   IF (OutRegs.ax AND &HFF) = &H1A THEN
```

```
      Code = OutRegs.bx AND &HFF
      SELECT CASE Code
         CASE 1
            GetVideoCard$ = "MDA"
         CASE 2
            GetVideoCard$ = "CGA"
         CASE 4 TO 5
            GetVideoCard$ = "EGA"
         CASE 6
            GetVideoCard$ = "PGA"
         CASE 7 TO 8
            GetVideoCard$ = "VGA"
         END SELECT
      EXIT FUNCTION
   ELSE
      InRegs.ax = &H1200
      InRegs.bx = &H10
      CALL INTERRUPT(&H10, InRegs, OutRegs)
      IF (OutRegs.bx AND &HFF) <> &H10 THEN
         GetVideoCard$ = "EGA"
         EXIT FUNCTION
         :
         :
```

In caso contrario, rimangono due sole ulteriori opzioni, CGA e MDA. Poiché la scheda MDA è in grado di gestire unicamente la modalità video BIOS 7, sarà sufficiente controllare la modalità video attiva: se è pari a 7, sarà stata installata una scheda MDA, in caso contrario, la scheda installata sarà una CGA.

```
FUNCTION GetVideoCard$

   DIM InRegs AS RegType, OutRegs AS RegType
   InRegs.ax = &H1A00
   CALL INTERRUPT(&H10, InRegs, OutRegs)
   IF (OutRegs.ax AND &HFF) = &H1A THEN
      Code = OutRegs.bx AND &HFF
      SELECT CASE Code
         CASE 1
            GetVideoCard$ = "MDA"
         CASE 2
            GetVideoCard$ = "CGA"
         CASE 4 TO 5
            GetVideoCard$ = "EGA"
         CASE 6
            GetVideoCard$ = "PGA"
         CASE 7 TO 8
            GetVideoCard$ = "VGA"
         END SELECT
      EXIT FUNCTION
```

```
    ELSE
       InRegs.ax = &H1200
       InRegs.bx = &H10
       CALL INTERRUPT(&H10, InRegs, OutRegs)
       IF (OutRegs.bx AND &HFF) <> &H10 THEN
          GetVideoCard$ = "EGA"
          EXIT FUNCTION
       ELSE
          InRegs.ax = &HF00
          CALL INTERRUPT(&H10, InRegs, OutRegs)
          IF (OutRegs.ax AND &HFF) = 7 THEN
             GetVideoCard$ = "MDA"
             EXIT FUNCTION
          ELSE
             GetVideoCard$ = "CGA"
             EXIT FUNCTION
          END IF
       END IF
    END IF
```

A questo punto la verifica di installazione scheda video è completa. Il listato 5.11 presenta completamente la funzione *GetVideoCard$( )*.

```
DECLARE FUNCTION GetVideoCard$ ()

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
OutRegs AS RegType)

    CLS
    PRINT "La scheda video installata è una "; GetVideoCard$

END
```

continua

Listato 5.11 *La funzione GetVideoCard$*

```
FUNCTION GetVideoCard$

REM Questa funzione restituisce una stringa di 3 lettere:
REM "MDA" (monochrome display adapter), "CGA" (color graphics
REM adapter), "EGA" (enhanced graphics adapter), "PGA"
REM (professional graphics adapter), "VGA" (variable graphics
REM array)

    DIM InRegs AS RegType, OutRegs AS RegType
    InRegs.ax = &H1A00
    CALL INTERRUPT(&H10, InRegs, OutRegs)
    IF (OutRegs.ax AND &HFF) = &H1A THEN
       Code = OutRegs.bx AND &HFF
       SELECT CASE Code
          CASE 1
             GetVideoCard$ = "MDA"
          CASE 2
             GetVideoCard$ = "CGA"
          CASE 4 TO 5
             GetVideoCard$ = "EGA"
          CASE 6
             GetVideoCard$ = "PGA"
          CASE 7 TO 8
             GetVideoCard$ = "VGA"
          END SELECT
       EXIT FUNCTION
    ELSE
       InRegs.ax = &H1200
       InRegs.bx = &H10
       CALL INTERRUPT(&H10, InRegs, OutRegs)
       IF (OutRegs.bx AND &HFF) <> &H10 THEN
          GetVideoCard$ = "EGA"
          EXIT FUNCTION
       ELSE
          InRegs.ax = &HF00
          CALL INTERRUPT(&H10, InRegs, OutRegs)
          IF (OutRegs.ax AND &HFF) = 7 THEN
             GetVideoCard$ = "MDA"
             EXIT FUNCTION
          ELSE
             GetVideoCard$ = "CGA"
             EXIT FUNCTION
          END IF
       END IF
    END IF
END FUNCTION
```

Listato 5.11 *La funzione GetVideoCard$.*

Per mettere in opera la funzione *GetVideoCard$( )* digitare quanto segue:

```
REM Esempio di GetVideoCard$( )
DECLARE FUNCTION GetVideoCard$ ( )
PRINT "La scheda video installata è una "; GetVideoCard$
```

Questo programma di esempio segnala quale scheda video è installata nel computer. Si passerà ora all'analisi di un'altra funzione che fornisce un'ulteriore informazione per l'ambiente grafico: la risoluzione verticale e orizzontale del monitor.

CONTROLLO DELLA RISOLUZIONE VERTICALE E ORIZZONTALE DEL VIDEO

Lo scopo dell'ultima funzione di questo capitolo (*GetXYRanges%(XRange%, YRange%)*) è quello di controllare l'ambiente grafico prima di impostarlo. La funzione che verrà presentata determinerà il numero di pixel gestibili dal video, nella modalità attiva, sia in orizzontale che in verticale. Questa funzione produce i seguenti risultati:

Nome	*Descrizione*
XRange%	Numero di pixel orizzontali nella modalità video attiva.
YRange%	Numero di pixel verticali nella modalità video attiva.
GetXYRanges%	*INTEGER 0* indica che non è attiva la modalità grafica; *INTEGER 1* indica l'intervallo *W* (o *Y*) di *XRange%* (oppure *YRange%*).

Si noti che vi è sempre la possibilità che lo schermo non sia in modalità grafica, nel qual caso il conteggio dei pixel non ha nessuna utilità e *GetXYRanges%( )* dovrebbe restituire un valore pari a *0*.
Il listato 5.12 presenta l'intera funzione (si osservi che essa è composta unicamente da una singola istruzione *SELECT CASE*, basata sulla modalità video del BIOS).

```
DECLARE FUNCTION GetXYRanges% (XRange%, YRange%)
REM Questa funzione restituisce il la risoluzione x e y dei pixel
REM per la modalità grafica attiva.

TYPE RegType
    ax      AS INTEGER
    bx      AS INTEGER
    cx      AS INTEGER
```

continua

Listato 5.12 *La funzione GetXYRanges%( )*

```
    dx      AS INTEGER
    bp      AS INTEGER
    si      AS INTEGER
    di      AS INTEGER
    flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

SCREEN 1
Check% = GetXYRanges% (XR%, YR%)
PRINT "X Range: ",XR%
PRINT "Y Range: ",YR%
END

FUNCTION GetXYRanges% (XRange%, YRange%)
    DIM InRegs AS RegType, OutRegs AS RegType
    InRegs.ax = &HF00
    CALL INTERRUPT(&H10, InRegs, OutRegs)
    Mode = (&HFF AND OutRegs.ax)
    GetXYRanges% = 1
    SELECT CASE Mode
       CASE 4, 5, 13, 19
          XRange% = 320
          YRange% = 200
       CASE 6, 10, 14
          XRange% = 640
          YRange% = 200
       CASE 8
          XRange% = 160
          YRange% = 200
       CASE 9
          XRange% = 320
          YRange% = 200
       CASE 15, 16
          XRange% = 640
          YRange% = 350
       CASE 17, 18
          XRange% = 640
          YRange% = 480
       CASE ELSE
          GetXYRanges% = 0
    END SELECT

END FUNCTION
```

Listato 5.12 *La funzione GetXYRanges%( ).*

In questo listato non vi è nulla di particolare in quanto ogni elemento è stato impostato in relazione alla modalità video del BIOS. Di seguito viene riportato come utilizzare la funzione *GetXYRanges%( )*.

```
REM Esempio di GetXYRanges%

DECLARE FUNCTION GetXYRanges(XRange%, YRange%)

SCREEN 2
Check% GetXYRanges%(XRange%, YRange%)

LOCATE 1, 1
IF Check% < > 0 THEN
   PRINT "Gli intervalli X e Y sono: ",XRange%, YRange%
ENDIF
```

Si noti che viene eseguito il controllo di *GetXYRanges%* (per controllare che il valore restituito non sia pari a *0*) per accertarsi che *XRange%* e *YRange%* contengano valori ammissibili.

CONCLUSIONE

In questo capitolo si è sviluppato un programma di tipo *paint*, si è costruito e visualizzato un orologio, si è animato un piccolo missile, si è rappresentato graficamente il consumo del burro in quattro diverse regioni geografiche e si è analizzato come controllare l'ambiente grafico.
La grafica, se correttamente gestita, può indubbiamente aggiungere ai programmi che verranno creato un'aspetto davvero professionale.
Si è pronti ora a procedere con un tema che è caro a molti programmatori: la realizzazione di un database. Nel capitolo 6 si vedrà che il BASIC PDS è in grado di fornire validi strumenti per la programmazione di database.

CAPITOLO 6

DATABASE

In questo capitolo esploreremo il sistema ISAM (Indexed Sequential Access Method) del Basic Professional Development System. Conviene leggere queste pagine anche se non si possiede il PDS, se non altro per avere un'idea di quello che si ha a disposizione, in particolare se si è interessati ai database. Il sistema ISAM offre un modo semplice di impostare un programma di database dal Basic (e basta il fatto che esista questo sistema nel PDS a dimostrare quanto profondo sia l'impegno della Microsoft per questo linguaggio di programmazione). Si tratta di un sistema serio, pensato per applicazioni serie.

Nota: *Il sistema ISAM costituisce uno dei motivi principali per cui i programmatori passano al PDS.*

Che cos'è dunque ISAM? Detto in poche parole, è un metodo per modificare l'ordine apparente dei record in un file. Per esempio, si potrebbe desiderare di avere l'elenco degli amici ordinato per altezza o per età. Per questo, è possibile che si sia creato un nuovo tipo di dati in cui conservare le informazioni relative a ciascuna persona:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE
```

Ciascuna variabile in questo tipo è un campo, nella terminologia dei database, e, presi tutti insieme, questi campi costituiscono un record. Dopo aver definito un tipo di dati, si si può definire una variabile di questo tipo, con il nome MyFriend, in questo modo:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend
```

Poi si possono riempire con i valori opportuni tutti i campi del record, per esempio:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170     'Centimetri
MyFriend.Residenza = "Redlands"
```

Si può anche creare un file con questi record, come nel caso visualizzato in figura 6.1: questo file è un database. I record che contiene vi figurano nell'ordine in cui sono stati inseriti, uno dopo l'altro, cioè nel cosiddetto *ordine di inserimento.*

Se lo trasformassimo in un file ISAM (cosa che dovremmo specificare al momento della creazione del file), il sistema ISAM aprirebbe automaticamente il file con un indice dei record, come si può vedere in figura 6.2.

L'indice non fa altro che elencare i numeri di record nell'ordine in cui sono stati inseriti nel file: nel sistema ISAM questo indice viene denominato indice NULL (figura 6.3). L'indice NULL è l'indice implicito dei record del file e viene sempre creato quando si apre un file ISAM per output e vi si inseriscono dei record.

Tuttavia, il punto chiave del sistema ISAM è la possibilità di creare anche altri indici. Per esempio si può dire al sistema ISAM di creare un indice dei record presenti nel file ordinato per età. Per richiamarlo facilmente, possiamo anche dare un nome a questo indice: per esempio, AmiciPerEta (figura 6.4). Si osservi che l'ordine in cui compaiono i record in questo indice è diverso da quello dell'indice NULL. Qui i record sono ordinati in termini di età crescente.

Per creare indici con ISAM si usa l'istruzione del Basic CREATEINDEX. Si può creare, in modo analogo, un altro indice sulla base dell'altezza delle persone nel database, indice che si può chiamare AmiciPerAltezza (figura 6.5).

Nome:	"Doug"	Record 1
Eta:	33	
Altezza:	170	
Residenza:	"Redlands"	
Nome:	"Ed"	Record 2
Eta:	44	
Altezza:	169	
Residenza:	"San Pedro"	
Nome:	"Margie"	Record 3
Eta:	67	
Altezza:	164	
Residenza:	"Los Angeles"	
Nome:	"Adam"	Record 4
Eta:	25	
Altezza:	171	
Residenza:	"Morgantown"	

Figura 6.1

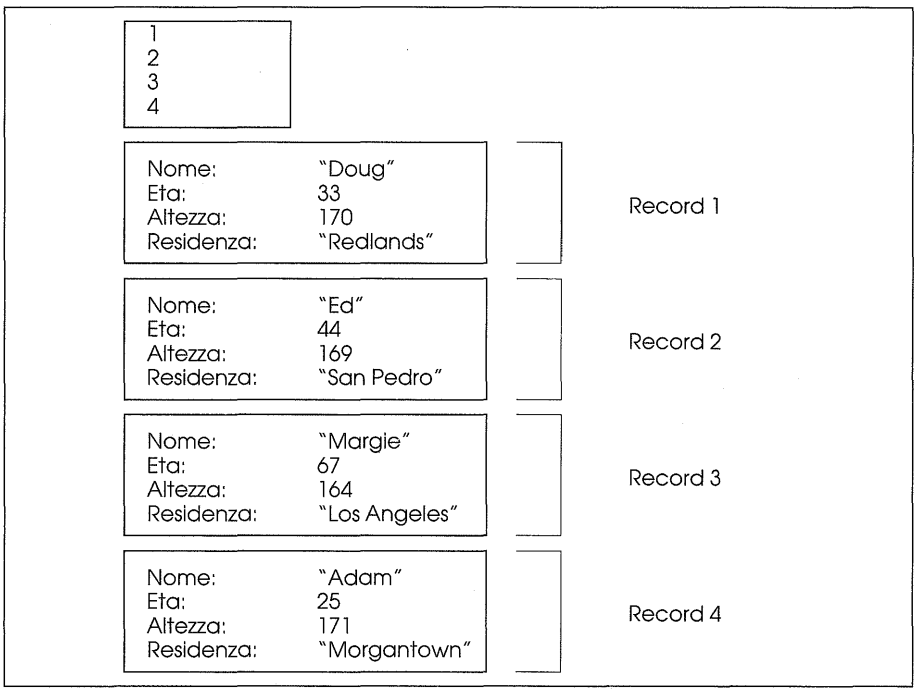

Figura 6.2

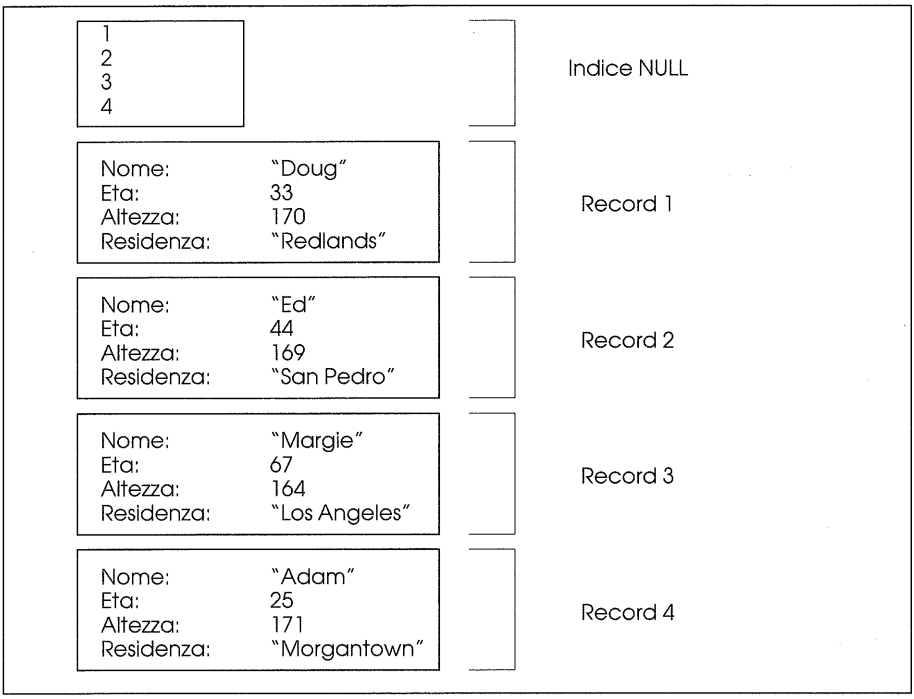

Figura 6.3

L'INDICE CORRENTE DEL DATABASE

Questi indici diventano parte del file (se ne occupa automaticamente ISAM all'atto della creazione di un nuovo indice). Un solo indice alla volta, tuttavia, può essere l'indice corrente. Tutte le volte poi che si cambia l'indice corrente, l'ordine apparente dei record nel file muta, in funzione dell'ordinamento del nuovo indice. Prima che venga creato un indice, l'indice corrente è NULL (figura 6-6). Se si chiede il primo record del file, si ottiene il record 1; il successivo sarà il record 2 e via di questo passo, seguendo semplicemente l'ordine di inserimento dei record. Si può stabilire quale debba essere l'indice corrente con l'istruzione SETINDEX. Per esempio, con SETINDEX si può fare in modo che l'indice corrente diventi AmiciPerEta (figura 6-7). Ora, quando si chiede il primo record del file, si ottiene quello che in realtà è il record 4, non il record 1 (Adam è il più giovane). Il record successivo sarà il record 1m seguito dal 2 e dal 3. In altre parole, ora il file appare ordinato in base all'età (si badi che il file non è stato fisicamente riordinato. Il sistema ISAM usa dei puntatori ai vari record e sono questi puntatori che vengono fisicamente riordinati, non i record stessi).

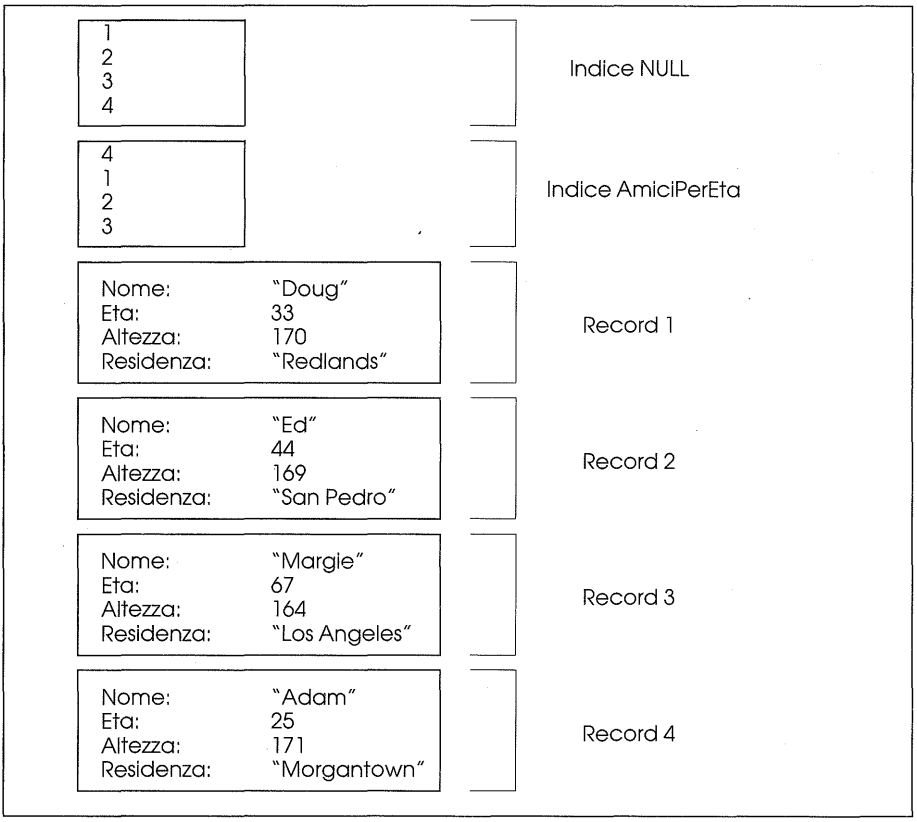

Figura 6.4

> ***Consiglio:*** Ordinare un file ordinando i puntatori ai record, anziché i record stessi, risparmia a ISAM il problema di spostare i dati relativi a ciascun record. Si tratta di un trucco utilizzato da tutti i sistemi professionali di database e il risultato è che ISAM può gestire i record a una velocità notevolmente superiore a qualsiasi altro metodo del Basic.

Analogamente, se si fosse scelto AmiciPerAltezza come indice corrente, il file apparirebbe ordinato in base all'altezza. Quando si recupera il primo record del file, allora, si ottiene il record della persona più bassa; il successivo sarà quello della persona seguente in ordine di altezza e così via fino alla persona più alta del file.

Ordinare i record in questo modo può essere molto utile. Per esempio, se si vuole escludere tutte le persone di età inferiore ai 30 anni, si ordina il file per età e si ottiene, record per record:

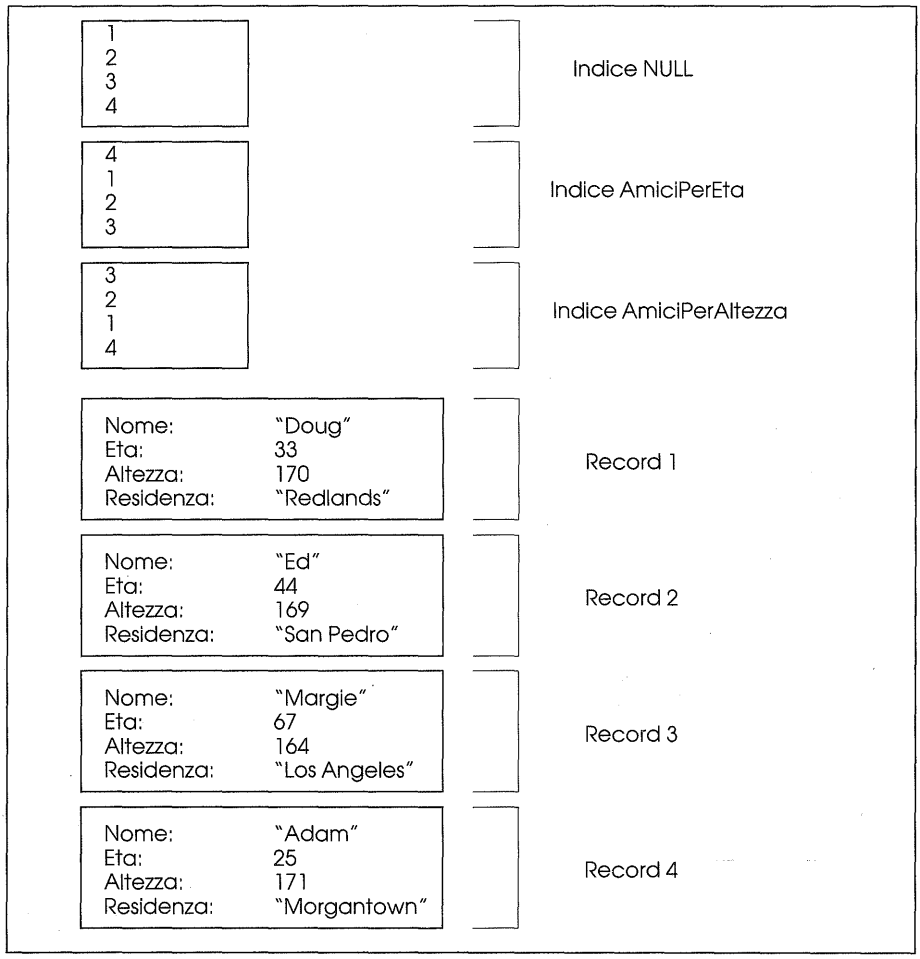

Figura 6.5

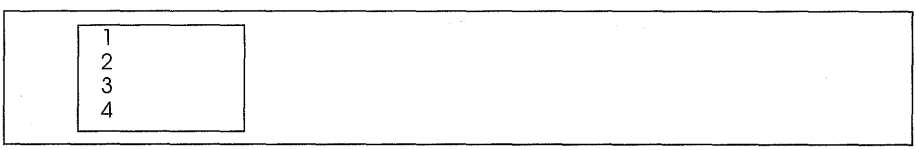

Figura 6.6

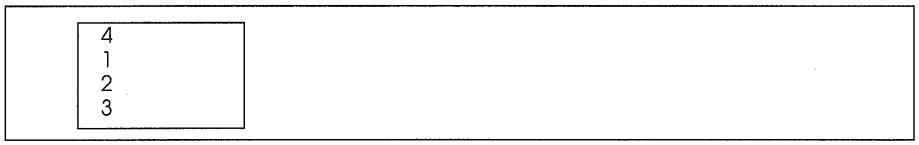

Figura 6.7

25 (Adam)
33 (Doug)
44 (Ed)
67 (Margie)

Poi si può esplorare il file finché non si trova per la prima volta un'età superiore ai 30 (cosa che si fa con l'istruzione SEEK, come vedremo), il che accade per Doug:

25 (Adam)
33 (Doug)
44 (Ed)
67 (Margie)

Poiché i record sono ordinati, si sa che tutti i record precedenti si riferiscono a persone di età inferiore ai trenta, perciò li si può scartare. Si conservano invece i dati successivi. Gestire i dati in modo simile costituisce la parte fondamentale di un programma di database. E ora si codificherà il tutto in BASIC.

UN PROGRAMMA DI DATABASE

Prima di scrivere effettivamente qualcosa in Basic, va detto che, per poter lavorare con ISAM, è necessaria un po' di preparazione.

Questa volta non è necessario creare o caricare nuove librerie Quick: bisogna invece caricare un programma TSR (Terminate and Stay Resident) che contiene il codice ISAM. Il programma si chiama PROISAMD.EXE e contiene tutto il codice necessario per eseguire i programmi di database.

Basta farlo girare una volta prima di usare qualsiasi routine ISAM e poi far partire QBX come al solito. Se il programma di database fosse DB.BAS, bisognerebbe battere QBX DB. Se si usa BC.EXE (versione 7.0 o successive) basta compilare e fare il link del programma per ottenere DB.EXE, poi far girare PROISAM.EXE e infine DB.EXE.

> ***Consiglio:*** Esiste un programma più breve di PROISAMD.EXE, e cioè PROISAM.EXE, che contiene tutto quel che serve per la maggior parte dei programmi ISAM. Questo programma più breve consente una minore occupazione di RAM, ma esclude alcune routine, tra le quali proprio CREATEINDEX.

Ora siamo pronti per un po' di programmazione. Il programma di database farà alcune delle cose di cui si è parlato e potremo anche usare la lista di amici che abbiamo sviluppato. Inizieremo con lo strutturare i record di dati con TYPE, per poi creare una variabile di quel tipo, con il nome MyFriend:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend
:
:
```

Ora si può creare il file ISAM. È sufficiente scrivere OPEN "FRIENDS.DAT" FOR ISAM (anziché aprirlo con altre opzioni del Basic come APPEND), in questo modo:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1
:
:
```

Si dia un'occhiata a questa istruzione OPEN. Si comincia con OPEN "FRIENDS.DAT" FOR ISAM, che è abbastanza chiaro, poiché si vuole il file sia un file ISAM. La parola chiave successiva, Friend, indica il tipo dei record che verranno usati, cosicché ISAM possa strutturare ciascun record del file. La parola chiave successiva, Pals è il nome di tabella e il numero (#1) è il numero del file, come di norma in Basic. Tutti i record di un particolare tipo di dati (come il tipo Friend) costituiscono una tabella nel file di database. Per esempio, quando si sarà riempito FRIENDS.DAT con tutti i nomi degli amici, tutti i record si troveranno in un'unica tabella, a cui si attribuirà il nome Pals (figura 6.8).

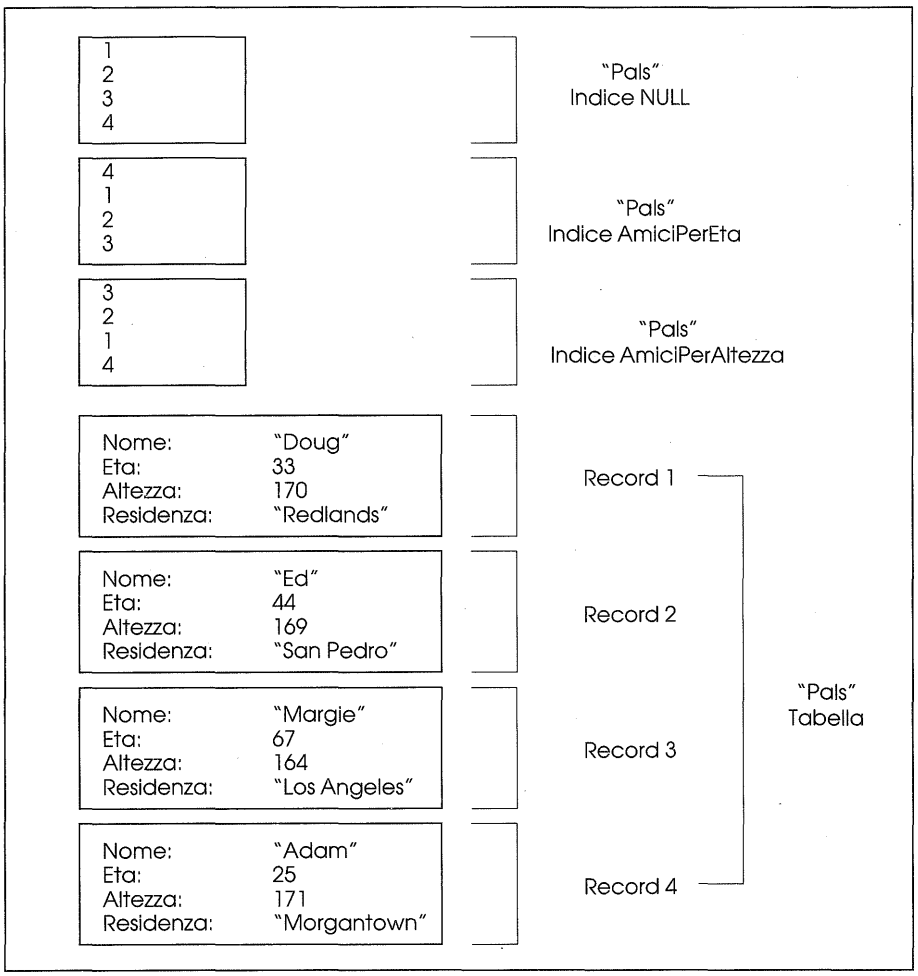

Figura 6.8

Si può avere nello stesso file anche un'altra tabella con record di tipo diverso (poniamo, per esempio, di tipo Enemy), che costituirebbero una nuova tabella, a cui si può dare il nome NotPals (figura 6.9).
Se si volesse lavorare con la tabella NotPals, si aprirebbe di nuovo *lo stesso* file, ma questa volta si dovrebbe specificare un diverso tipo record (Enemy) e un diverso numero di file:

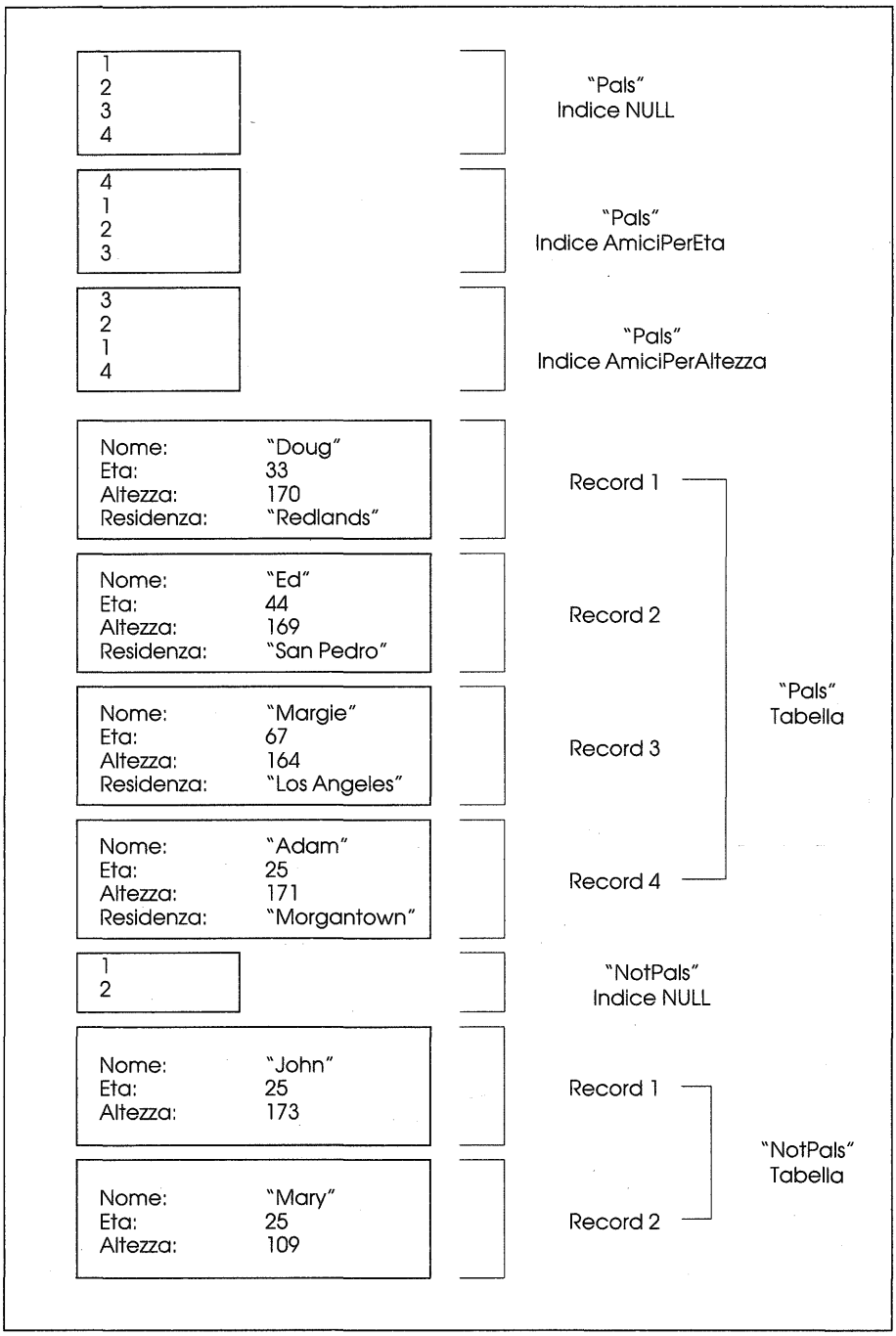
1
2
3
4
"Pals"
Indice NULL
4
1
2
3
"Pals"
Indice AmiciPerEta
3
2
1
4
"Pals"
Indice AmiciPerAltezza
Nome: "Doug"
Eta: 33
Altezza: 170
Residenza: "Redlands"
Record 1
Nome: "Ed"
Eta: 44
Altezza: 169
Residenza: "San Pedro"
Record 2
Nome: "Margie"
Eta: 67
Altezza: 164
Residenza: "Los Angeles"
Record 3
Nome: "Adam"
Eta: 25
Altezza: 171
Residenza: "Morgantown"
Record 4
"Pals"
Tabella
1
2
"NotPals"
Indice NULL
Nome: "John"
Eta: 25
Altezza: 173
Record 1
Nome: "Mary"
Eta: 25
Altezza: 109
Record 2
"NotPals"
Tabella

Figura 6.9

```
OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1
OPEN "FRIENDS.DAT" FOR ISAM Enemy "NotPals" AS #2
```

Quello che di norma nel BASIC in generale viene definito un numero di file (1 e 2, come sono stati aperti fin qui) in ISAM è in realtà un *numero di tabella.* Quando si creano indici, si cercano corrispondenze o si eseguono altre operazioni in ISAM, si deve specificare anche il numero di tabella.

> ***Consiglio:*** Il motivo per cui si possono avere più tabelle in ISAM è quello di evitare la necessità di database relazionali, dove si devono collegare record di file diversi. Qui si possono inserire nello stesso file tutti i dati, anche quelli che normalmente andrebbero in file distinti. I file ISAM, poi, possono essere enormi: il limite massimo è 128 megabyte.

In questo capitolo si lavorerà con una sola tabella (Pals) e il numero della tabella sarà 1:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1
:
:
```

Ora bisogna inserire i dati desiderati, record per record, in FRIENDS.DAT. Lo si può fare con l'istruzione INSERT, in questo modo:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170     'Centimetri
```

```
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169     'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164     'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend

MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171     'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend
:
:
```

È un procedimento molto semplice: basta compilare i campi di MyFriend con i dati desiderati, poi effettuare l'INSERT di MyFriend nel file FRIENDS.DAT (aperto come #1).
A questo punto il file di database FRIENDS.DAT contiene i quattro record caricati e ISAM ha impostato automaticamente l'indice NULL (figura 6.10).
Ora si può passare a creare gli altri due indici, AmiciPerEta e AmiciPerAltezza. A tale scopo si usa CREATEINDEX, a cui vanno passati il numero del file, il nome che si vuole attribuire all'indice, un parametro *unico%* e il nome del campo rispetto al quale si vuole l'ordinamento (Eta, in questo caso):

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170     'Centimetri
MyFriend.Residenza = "Redlands"
```

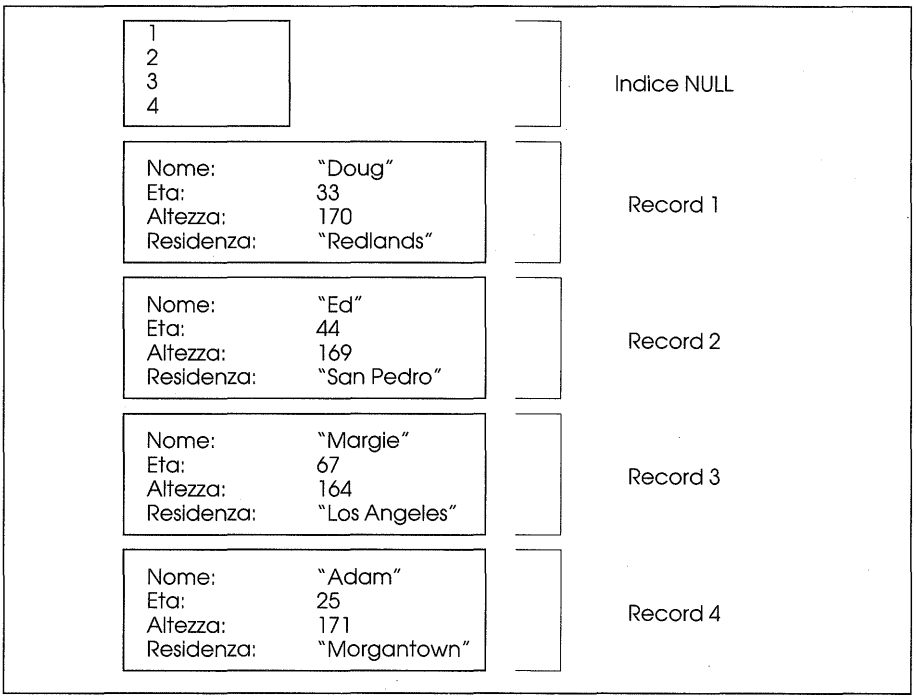

Figura 6.10

```
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169     'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164     'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend

MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171     'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend
```

```
PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"
:
:
```

Il campo unico% indica se ISAM ammetterà o meno l'inserimento di record che abbiano in questo campo un valore già esistente nel database. Per esempio, se si imposta unico% a 1, chiedendo perciò che ogni valore compaia una sola volta, e poi si cerca di inserire un record per Pete, la cui età è 33 (come per Doug), si genererebbe un errore intercettabile (si usa cioè ON ERROR GOTO). In questo capitolo non ci si preoccuperà dell'unicità dei record, perciò il parametro viene impostato a 0. Ora il database è come viene rappresentato in figura 6.11.
Si sa in quale ordine siano stati inseriti i record nel file; ora li si può ordinare in base all'età rendendo corrente l'indice AmiciPerEta e stampando il nome di ciascuno degli amici registrati. Si comincia con SETINDEX:

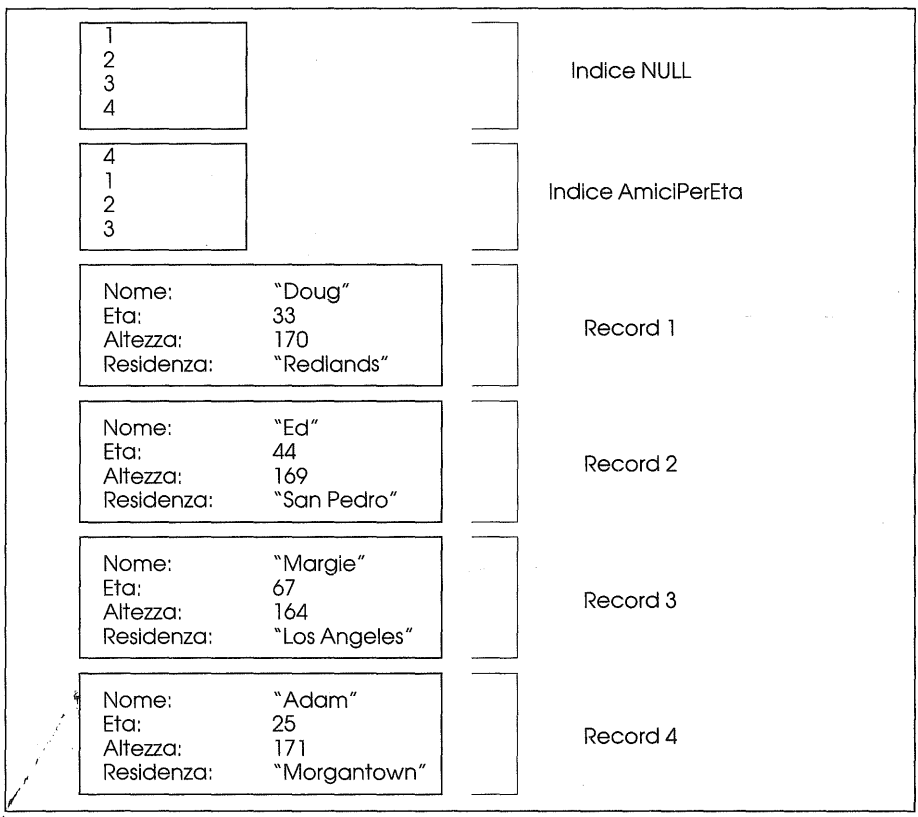

Figura 6.11

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170     'Centimetri
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169     'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164     'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend

MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171     'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend


PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"

SETINDEX 1, "AmiciPerEta"
:
:
```

Abbastanza facile: non si fa altro che fare di AmiciPerEta l'indice corrente per il file 1 (in realtà la tabella 1). Ora si possono stampare i nomi nel nuovo ordine.
Il sistema ISAM offre numerose istruzioni per la navigazione in un file di database. Ecco un elenco di quelli che risulteranno particolarmente utili:

```
MOVEFIRST NumeroFile
MOVELAST NumeroFile
MOVENEXT NumeroFile
MOVEPREVIOUS NumeroFile
EOF(NumeroFile)
BOF(NumeroFile)
```

Bisogna sempre ricordare che quel che nella documentazione di ISAM è indicato come NumeroFile (o FileNumber) è in effetti il numero delle tabelle. In altre parole, per quanto possa sembrare confuso, si possono avere per uno specifico file tanti numeri di file diversi quante sono le tabelle in quel file.

Le istruzioni MOVE portano, rispettivamente, al primo, all'ultimo, al successivo e al precedente elemento del file; EOF(NumeroFile) restituisce TRUE (cioè tutti i bit con valore 1: &HFFFF) se ci si trova alla fine della tabella; BOF(NumeroFile) restituisce TRUE se invece ci si trova all'inizio. In caso contrario, restituiscono FALSE (0).

Tenendo presente tutto questo, ecco come si può stampare il nuvo ordinamento dei record di FRIENDS.DAT, ora disposti in ordine per età:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170     'Centimetri
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169     'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164     'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend
```

```
MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171     'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend


PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"
SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)
:
:
```

Quello che si fa è spostarsi sul primo record del file:

```
MOVEFIRST 1
:
```

Poi si entra in un ciclo DO che percorre i singoli record del file utilizzando continuamente MOVENEXT finché non diventa diverso EOF(1):

```
MOVEFIRST 1

DO
    :
    :
    MOVENEXT 1
LOOP WHILE NOT EOF(1)
```

Quando si è all'inizio del file, il primo record dell'indice corrente è il *record corrente.* Quando si usa MOVENEXT, diventa corrente il record successivo in quell'indice. Si può esaminare il record corrente recuperandolo, con RETRIEVE, e inserendolo in una variabile di tipo Friend, come MyFriend:

```
MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    :
    MOVENEXT 1
LOOP WHILE NOT EOF(1)
```

RETRIEVE 1, MyFriend legge il record corrente dal file di database e lo colloca in MyFriend. Questo significa che a quel punto si può stampare il nome della persona nel record corrente con un'istruzione PRINT di questo tipo:

```
MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)
```

E in questo modo si passa in ciclo per tutti i record ordinati di un file ISAM.
Si può creare anche il secondo indice, AmiciPerAltezza, in questo modo:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170     'Centimetri
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169     'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164     'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend

MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171     'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend
```

```
PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)


PRINT "Sto ordinando per altezza..."

CREATEINDEX 1, "AmiciPerAltezza", 0, "Altezza"
:
:
```

La struttura è esattamente identica a quella usata per la creazione di AmiciPerEta, solo che ora si ordina in base al campo Altezza e si dà un diverso nome all'indice. A questo punto, il database appare come in figura 6.12.
Si può fare in modo che AmiciPerEta diventi l'indice corrente e poi si possono stampare i record in quell'ordine, come si è fatto per AmiciPerEta:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170    'Centimetri
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169    'Centimetri
```

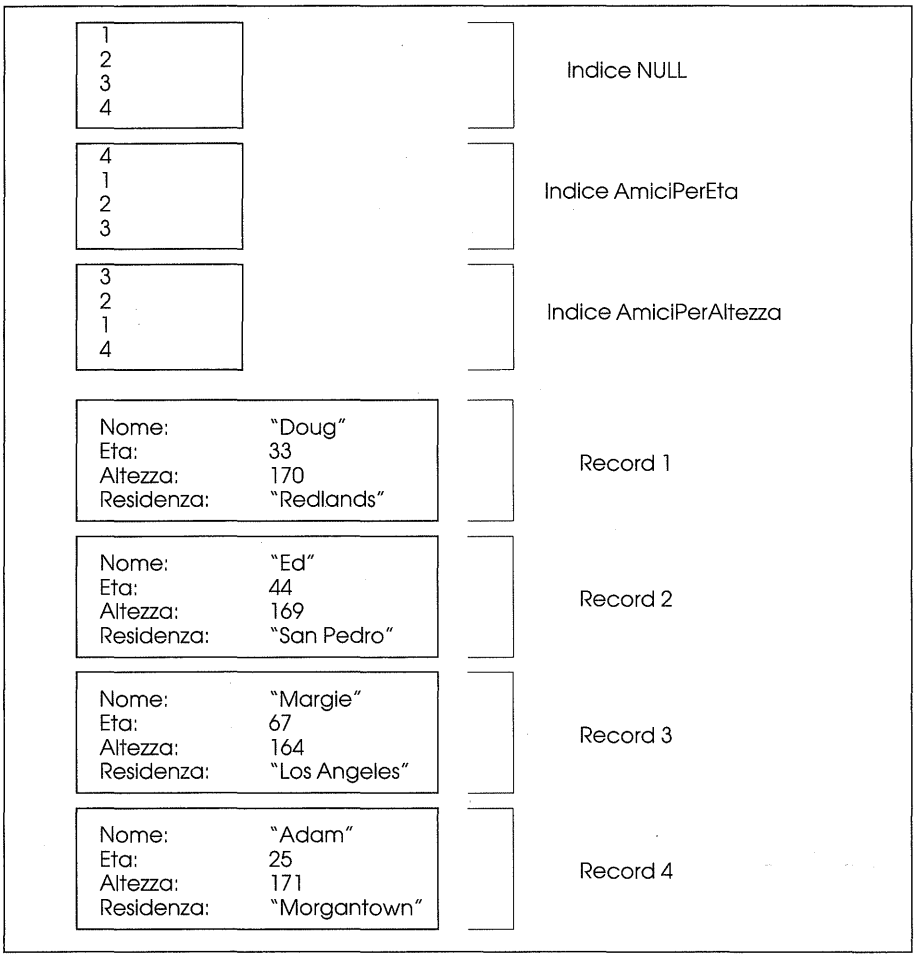

Figura 6.12

```
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164     'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend
MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171     'Centimetri
```

```
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend


PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)


PRINT "Sto ordinando per altezza..."

CREATEINDEX 1, "AmiciPerAltezza", 0, "Altezza"

SETINDEX 1, "AmiciPerAltezza"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)
:
:
```

E questo è tutto per il riordinamento dell'intero file. Ora si può cominciare a esplorare il file alla ricerca di valori particolari.

ALLA RICERCA DEI RECORD

Il lavoro con un database consiste in gran parte nel riuscire a trovare rapidamente i record che soddisfano un determinato criterio. Per esempio, può capitare di dover cercare una persona che abbia 33 anni.
Per eseguire operazioni come questa (e altre simili), ISAM fornisce le istruzioni SEEK: SEEKGT, SEEKGE e SEEKEQ. Usarle non è difficile. Se per esempio si volesse trovare una persona di 33 anni nella tabella 1 del file appena impostato, si dovrebbe rendere

indice corrente AmiciPerEta (in modo che ISAM sappia a quale campo si è interessati) e poi si dovrebbe utilizzare l'istruzione SEEKEQ 1, 33. Ecco il procedimento codificato:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170    'Centimetri
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169    'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164    'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend

MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171    'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend


PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"

SETINDEX 1, "AmiciPerEta"
MOVEFIRST 1
```

```
DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)


PRINT "Sto ordinando per altezza..."

CREATEINDEX 1, "AmiciPerAltezza", 0, "Altezza"

SETINDEX 1, "AmiciPerAltezza"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)

PRINT "Sto cercando una persona di 33 anni..."

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

SEEKEQ 1, 33
:
:
```

> ***Consiglio:*** Si possono preparare procedure proprie in ISAM utilizzando MOVE, SEEK e EOF.

Se non viene trovato alcun record che soddisfi il criterio, viene impostato a 1 EOF(1). Se EOF(1) non è vero, invece, è stato trovato un record che soddisfa il requisito e sarà ora il record corrente, che quindi si può evidenziare e stampare con RETRIEVE:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend
```

```
OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170    'Centimetri
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169    'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164    'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend

MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171    'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend

PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)


PRINT "Sto ordinando per altezza..."

CREATEINDEX 1, "AmiciPerAltezza", 0, "Altezza"

SETINDEX 1, "AmiciPerAltezza"

MOVEFIRST 1
```

```
DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)

PRINT "Sto cercando una persona di 33 anni..."

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

SEEKEQ 1, 33

IF EOF(1) THEN
        PRINT "Nessun record corrispondente."
ELSE
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
END IF
:
:
```

In modo analogo, si può cercare una persona che sia alta più 184 centimetri, utilizzando SEEKGT:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170     'Centimetri
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169     'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend
```

```
MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164     'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend

MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171     'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend

PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)

PRINT "Sto ordinando per altezza..."

CREATEINDEX 1, "AmiciPerAltezza", 0, "Altezza"

SETINDEX 1, "AmiciPerAltezza"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)


PRINT "Sto cercando una persona di 33 anni..."

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

SEEKEQ 1, 33
```

```
IF EOF(1) THEN
        PRINT "Nessun record corrispondente."
ELSE
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
END IF

PRINT "Sto cercando qualcuno che sia alto più di 184 cm."

SETINDEX 1, "AmiciPerAltezza"

MOVEFIRST 1

SEEKGT 1, 184

IF EOF(1) THEN
        PRINT "Nessun record corrispondente."
ELSE
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
END IF
:
:
```

Poiché nessun record soddisfa il criterio, EOF(1) è TRUE e viene stampato "Nessun record corrispondente."
ISAM offre vari altri metodi per la gestione dei record, oltre a consentire, come abbiamo visto, l'inserimento: è possibile cancellare record o modificare i valori dei loro campi. Vediamo, per cominciare, il procedimento di cancellazione.

CANCELLAZIONE DI RECORD

Si supponga di voler cancellare tutti i record relativi ad amici che abbiano superato la quarantina. Si possono cercare questi record rendendo corrente l'indice AmiciPerEta (con SETINDEX), per poi usare l'istruzione SEEKGT 1, 40.
Se si trova qualche record che soddisfa il criterio (se, cioè, EOF(1) è FALSE), basta cancellarli con l'istruzione DELETE 1, che cancella il record corrente nella tabella 1.
Dopo aver eliminato tutti i record che soddisfano il criterio, si può stampare l'elenco dei record restanti:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE
```

```
DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170    'Centimetri
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169    'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164    'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend

MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171    'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend

PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)

PRINT "Sto ordinando per altezza..."

CREATEINDEX 1, "AmiciPerAltezza", 0, "Altezza"

SETINDEX 1, "AmiciPerAltezza"
```

```
MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)

PRINT "Sto cercando una persona di 33 anni..."

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

SEEKEQ 1, 33

IF EOF(1) THEN
        PRINT "Nessun record corrispondente."
ELSE
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
END IF

PRINT "Sto cercando qualcuno che sia alto più di 184 cm."

SETINDEX 1, "AmiciPerAltezza"

MOVEFIRST 1

SEEKGT 1, 184

IF EOF(1) THEN
        PRINT "Nessun record corrispondente."
ELSE
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
END IF

PRINT "Cancello tutti quelli che hanno più di 40 anni..."

SETINDEX 1, "AmiciPerEta"

DO
    SEEKGT 1, 40

    IF NOT EOF(1) THEN
        RETRIEVE 1, MyFriend
        DELETE 1
```

```
    END IF

LOOP UNTIL EOF(1)

PRINT "Ecco quelli che restano..."

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)
:
:
```

Come si può notare, cancellare record è facile, ma bisogna stare attenti, perché è impossibile fare marcia indietro. Inoltre, non bisogna aspettarsi che il file ISAM (FRIENDS.DAT, in questo caso) diventi per forza più piccolo quando si cancellando record. Tanto per cominciare, le dimensioni minime di un file ISAM sono 64K, di cui 32K sono per la gestione e 32K sono lo spazio per i record.
Quando lo spazio per i dati è tutto utilizzato, il sistema ISAM aggiunge ulteriore spazio a blocchi di 32K (questo perché i suoi algoritmi interni di ricerca funzionano meglio con blocchi di dati di queste dimensioni). Quando si cancella un record, il file non viene compattato: viene invece semplicemente reso disponibile lo spazio di quel record. In futuro in quello spazio potrà essere scritto un nuovo record.

> ***Consiglio:*** Se si cancellano molti record e si vuole effettivamente compattare un file ISAM, si può utilizzare il programma di utility ISAMPACK, ma bisogna tener presente che lavora a blocchi di 32K e di conseguenza può compattare un file solo se sono stati cancellati più di 32K di dati.

Oltre a cancellare record, è possibile modificare i singoli campi di ciascun record, grazie all'istruzione UPDATE.

AGGIORNAMENTO DI FILE ISAM

Si supponga che sia il compleanno di Doug, e che quindi la sua età passi da 33 a 34 anni; si potrebbe cancellare il suo record in FRIENDS.DAT e poi inserirne uno nuovo con la sua età corretta, ma il sistema ISAM consente un metodo di gran lunga più facile.
Per aggiornare i record, si può utilizzare l'istruzione UPDATE. In questo caso, tutto quel che si deve fare è rendere corrente il record di Doug, leggerlo in una variabile

di tipo Friend (per esempio, MyFriend) con RETRIEVE, cambiare in 34 il valore del campo Eta, si potrà poi usare l'istruzione UPDATE 1, MyFriend per aggiornare FRIENDS.DAT. Questa istruzione aggiorna il record corrente nel file 1 utilizzando i valori in MyFriend. Il procedimento nel suo complesso è il seguente:

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE

DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170     'Centimetri
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169     'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164     'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend

MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171     'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend

PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1
```

```
DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)

PRINT "Sto ordinando per altezza..."

CREATEINDEX 1, "AmiciPerAltezza", 0, "Altezza"

SETINDEX 1, "AmiciPerAltezza"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)

PRINT "Sto cercando una persona di 33 anni..."

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

SEEKEQ 1, 33

IF EOF(1) THEN
        PRINT "Nessun record corrispondente."
ELSE
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
END IF


PRINT "Sto cercando qualcuno che sia alto più di 184 cm."

SETINDEX 1, "AmiciPerAltezza"

MOVEFIRST 1

SEEKGT 1, 184

IF EOF(1) THEN
        PRINT "Nessun record corrispondente."
ELSE
    RETRIEVE 1, MyFriend
```

```
    PRINT MyFriend.Nome
END IF

PRINT "Cancello tutti quelli che hanno più di 40 anni..."

SETINDEX 1, "AmiciPerEta"

DO
    SEEKGT 1, 40

    IF NOT EOF(1) THEN
        RETRIEVE 1, MyFriend
        DELETE 1
    END IF

LOOP UNTIL EOF(1)

PRINT "Ecco quelli che restano..."

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)

PRINT "Cambio in 34 l'età di Doug..."

CREATEINDEX 1, "AmiciPerNome", 0, "Nome"

SETINDEX 1, "AmiciPerNome"

SEEKEQ 1, "Doug"

RETRIEVE 1, MyFriend
MyFriend.Eta = 34

UPDATE 1, MyFriend
:
:
```

Poi si possono stampare nome ed età di tutti gli amici nel file, per controllare che il cambiamento sia stato effettuato correttamente.
Ed ecco il programma finale.

```
TYPE Friend
        Nome AS STRING * 50
        Eta AS INTEGER
        Altezza AS INTEGER
        Residenza AS STRING * 50
END TYPE
DIM MyFriend AS Friend

OPEN "FRIENDS.DAT" FOR ISAM Friend "Pals" AS #1

MyFriend.Nome = "Doug"
MyFriend.Eta = 33
MyFriend.Altezza = 170     'Centimetri
MyFriend.Residenza = "Redlands"
INSERT 1, MyFriend

MyFriend.Nome = "Ed"
MyFriend.Eta = 44
MyFriend.Altezza = 169     'Centimetri
MyFriend.Residenza = "San Pedro"
INSERT 1, MyFriend

MyFriend.Nome = "Margie"
MyFriend.Eta = 67
MyFriend.Altezza = 164     'Centimetri
MyFriend.Residenza = "Los Angeles"
INSERT 1, MyFriend

MyFriend.Nome = "Adam"
MyFriend.Eta = 25
MyFriend.Altezza = 171     'Centimetri
MyFriend.Residenza = "Morgantown"
INSERT 1, MyFriend

PRINT "Sto ordinando per età..."

CREATEINDEX 1, "AmiciPerEta", 0, "Eta"

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
```

continua

Listato 6.1 *Un programma di database che usa ISAM*

```
LOOP WHILE NOT EOF(1)

PRINT "Sto ordinando per altezza..."

CREATEINDEX 1, "AmiciPerAltezza", 0, "Altezza"

SETINDEX 1, "AmiciPerAltezza"

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)

PRINT "Sto cercando una persona di 33 anni..."

SETINDEX 1, "AmiciPerEta"

MOVEFIRST 1

SEEKEQ 1, 33

IF EOF(1) THEN
        PRINT "Nessun record corrispondente."
ELSE
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
END IF

PRINT "Sto cercando qualcuno che sia alto più di 184 cm."

SETINDEX 1, "AmiciPerAltezza"

MOVEFIRST 1

SEEKGT 1, 184

IF EOF(1) THEN
        PRINT "Nessun record corrispondente."
ELSE
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
END IF
PRINT "Cancello tutti quelli che hanno più di 40 anni..."
```

continua

Listato 6.1 *Un programma di database che usa ISAM*

```
SETINDEX 1, "AmiciPerEta"

DO
    SEEKGT 1, 40

    IF NOT EOF(1) THEN
        RETRIEVE 1, MyFriend
        DELETE 1
    END IF

LOOP UNTIL EOF(1)

PRINT "Ecco quelli che restano..."

MOVEFIRST 1

DO
    RETRIEVE 1, MyFriend
    PRINT MyFriend.Nome
    MOVENEXT 1
LOOP WHILE NOT EOF(1)


PRINT "Cambio in 34 l'età di Doug..."

CREATEINDEX 1, "AmiciPerNome", 0, "Nome"

SETINDEX 1, "AmiciPerNome"

SEEKEQ 1, "Doug"

RETRIEVE 1, MyFriend
MyFriend.Eta = 34

UPDATE 1, MyFriend

SetIndex 1, "AmiciPerEta"

DO
    RETRIEVE 1, MyFriend
    PRINT "L'età di ";MyFriend.Nome;" è: ";MyFriend.Eta;" anni"
    MOVENEXT 1
LOOP WHILE NOT EOF(1)

CLOSE #1
```

Listato 6.1 *Un programma di database che usa ISAM.*

Alla fine, si noti che si chiude la tabella (e di conseguenza anche il file, perché contiene una sola tabella) con CLOSE 1.
E questo è tutto per il nostro database: non è proprio quel che si definirebbe un database di uso generale (non ha alcuna interfaccia utente), ma ci ha consentito di esplorare il sistema ISAM. Ora è tempo di passare ad altri metodi per la manipolazione dei dati in BASIC.

CAPITOLO 7

TECNICHE AVANZATE DI GESTIONE E ORDINAMENTO DEI DATI

In questo capitolo verranno esplorate pressoché tutte le modalità di organizzazione dei dati in Basic (e ne aggiungeremo qualcuna di nostra). Organizzare i dati in modo che sia facile accedervi può essere determinante, nello sviluppo dei programmi, per ottenere una buona velocità sia nella codifica, sia nell'esecuzione. In effetti, organizzare bene i dati sin dall'inizio può voler dire qualcosa di più che essere a metà dell'opera nella scrittura di un programma.

Verranno analizzati i metodi più utili per memorizzare i dati, in particolare matrici, strutture di dati, liste concatenate, buffer circolari e alberi binari. Chi programma (e in particolare i programmatori più avanzati) deve conoscere bene questi metodi comuni di organizzazione dei dati, per non reinventare continuamente l'acqua calda.

Alla fine del capitolo, poi, si vedranno due metodi di ordinamento veloce per ottenere il massimo dai dati, e un algoritmo di ricerca veloce per la ricerca in matrici ordinate.

LE VARIABILI DEL BASIC

Il metodo più semplice di organizzazione dei dati consiste nel memorizzarli in semplici variabili. Questi sono i tipi standard utilizzati in Basic.

Tipo	*Simbolo*	*Byte*	*Intervallo*
INTEGER	%	2	da -32768 a 32767
LONG	&	4	da -2147483648 a 2147483647
SINGLE	!	4	da -3.402823E38 a -1.40129E-45
DOUBLE	#	8	da -1.79769313486232D308 a -2.2250738585072D-308
CURRENCY	@	8	da -$922337203685477.5808 a $922337203685477.5807
STRING	$	32K	le stringhe possono essere lunghe fino a 32K caratteri (byte)

Si sono già visti già tutti questi tipi, salvo forse il nuovo tipo CURRENCY (valuta) disponibile nel Basic PDS, che può essere utilizzato per memorizzare somme di danaro. Ecco un esempio (i risultati sono stampati con arrotondamento al centesimo):

```
Saldo@ = 6000000.00
Affitto@ = 775000.00
Alimentari@ = 124000.50
Fatture@ = 513000.72

Saldo@ = Saldo@ - Affitto@
Saldo@ = Saldo@ - Alimentari@
Saldo@ = Saldo@ - Fatture@

PRINT "In banca restano Lire"; Saldo@
```

Questo esempio visualizza quanto rimane sul conto in banca dopo aver pagato affitto e varie altre spese. Si può pensare una variabile CURRENCY come un intero molto lungo con quattro decimali aggiunti (anche se gli ultimi due sono solo per la precisione interna). Si vedrà ancora il tipo CURRENCY nel Capitolo 11.

***Nota**: Si noti che il tipo valuta può rappresentare numeri oltre 400.000 volte più grandi degli interi LONG con la precisione degli interi. Per questo alcuni programmatori hanno smesso di usare il tipo LONG e usano al suo posto il tipo CURRENCY.*

L'ISTRUZIONE DATA

Il passo successivo nella gestione dei dati è l'istruzione DATA. Quasi tutti i programmatori in Basic la conoscono già, ma ecco un esempio in cui si calcolano la somma e il prodotto dei numeri da 1 a 10, memorizzate poi in un'istruzione DATA:

```
Sum& = 0
Product& = 1

FOR i = 1 TO 10
   READ Number&
   Sum& = Sum& + Number&
NEXT i

PRINT "La somma dei dati è:"; Sum&

RESTORE       'Reinizializza

FOR i = 1 TO 10
   READ Number&
   Product& = Product& * Number&
NEXT i

PRINT "Il prodotto dei dati è: "; Product&

DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

In questo caso, i dati sono definiti con l'istruzione DATA:

```
DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Vengono letti con la READ (che legge un numero dall'istruzione DATA e poi passa al numero successivo per la successiva operazione di lettura):

```
FOR i = 1 TO 10
   READ Number&
   Sum& = Sum& + Number&
NEXT i
```

Poi si può ripartire dall'inizio della lista, con RESTORE:

```
RESTORE       'Reinizializza

FOR i = 1 TO 10
   READ Number&
   Product& = Product& * Number&
NEXT i
```

Questo è un modo di gestire i dati. Ovviamente, quando si vogliono inserire dati numerici in un computer, uno degli obiettivi è quello di organizzarli, e il metodo più comune è quello che vedremo nel prossimo passo verso una organizzazione dei dati più raffinata, cioè le matrici.

MATRICI

Si sono già viste le matrici, come quella nel listato 7.1.

```
DIM Array(10, 2) AS CURRENCY
REM Inserisce in Array(n,1) le vendite del giorno:
Array(1, 1) = 10000.00
Array(2, 1) = 53000.00
Array(3, 1) = 7000.00
Array(4, 1) = 9670.00
Array(5, 1) = 8790.00
Array(6, 1) = 14000.00
Array(7, 1) = 9100.00
Array(8, 1) = 12700.00
Array(9, 1) = 1000.00
Array(10, 1) = 5400.00

REM Inserisce in Array(n, 2) le vendite del giorno precedente:
Array(1, 2) = 9670.00
Array(2, 2) = 3500.00
Array(3, 2) = 8970.00
Array(4, 2) = 10000.00
Array(5, 2) = 78000.00
Array(6, 2) = 17000.00
Array(7, 2) = 91360.00
Array(8, 2) = 12730.00
Array(9, 2) = 16120.00
Array(10, 2) = 7980.00

PRINT "     VENDITE (in Lire)"
PRINT "   Ieri            Oggi   "
PRINT "----------      ----------"
FOR i = 1 TO 10
    PRINT Array(i, 1), Array(i, 2)
NEXT i
PRINT "----------      ----------"

Sum1@ = 0
Sum2@ = 0
FOR i = 1 TO 10
    Sum1@ = Sum1@ + Array(i, 1)
    Sum2@ = Sum2@ + Array(i, 2)
NEXT i
PRINT Sum1@, Sum2@; " = Totale"
```

Listato 7.1 *Un esempio di matrice.*

In questo programma, si imposta una matrice di 10 righe e 2 colonne, destinata a contenere i valori di vendita degli ultimi due giorni:

```
DIM Array(10, 2) AS CURRENCY
REM Inserisce in Array(n,1) le vendite del giorno:
Array(1, 1) = 10000.00
Array(2, 1) = 53000.00
Array(3, 1) = 7000.00
Array(4, 1) = 9670.00
Array(5, 1) = 8790.00
Array(6, 1) = 14000.00
Array(7, 1) = 9100.00
Array(8, 1) = 12700.00
Array(9, 1) = 1000.00
Array(10, 1) = 5400.00

REM Inserisce in Array(n, 2) le vendite del giorno precedente:
Array(1, 2) = 9670.00
Array(2, 2) = 3500.00
Array(3, 2) = 8970.00
Array(4, 2) = 10000.00
Array(5, 2) = 78000.00
Array(6, 2) = 17000.00
Array(7, 2) = 91360.00
Array(8, 2) = 12730.00
Array(9, 2) = 16120.00
Array(10, 2) = 7980.00
        :
        :
```

La figura 7-1 visualizza la matrice risultante:

	Col 1	Col 2
Riga 1 →	10000.00	9670.00
Riga 2 →	53000.00	3500.00
	7000.00	8970.00
	9670.00	10000.00
	8790.00	78.000
	14000.00	17000.00
	9100.00	91360.00
	12700.00	12730.00
	1000.00	16120.00

Figura 7.1

Ora si può raggiungere la colonna delle vendite di ciascun giorno semplicemente modificando l'indice della colonna. In questo formato, si possono eseguire operazioni in parallelo su insiemi analoghi di dati, per esempio sommare i valori nelle singole colonne per avere i totali di ciascun giorno, come nell'esempio:

```
DIM Array(10, 2) AS CURRENCY

REM Inserisce in Array(n,1) le vendite del giorno:

Array(1, 1) = 10000.00
Array(2, 1) = 53000.00
Array(3, 1) = 7000.00
Array(4, 1) = 9670.00
Array(5, 1) = 8790.00
Array(6, 1) = 14000.00
Array(7, 1) = 9100.00
Array(8, 1) = 12700.00
Array(9, 1) = 1000.00
Array(10, 1) = 5400.00

REM Inserisce in Array(n, 2) le vendite del giorno precedente:
Array(1, 2) = 9670.00
Array(2, 2) = 3500.00
Array(3, 2) = 8970.00
Array(4, 2) = 10000.00
Array(5, 2) = 78000.00
Array(6, 2) = 17000.00
Array(7, 2) = 91360.00
Array(8, 2) = 12730.00
Array(9, 2) = 16120.00
Array(10, 2) = 7980.00

PRINT "     VENDITE (in Lire)"
PRINT "   Ieri              Oggi   "
PRINT "----------       ----------"
FOR i = 1 TO 10
   PRINT Array(i, 1), Array(i, 2)
NEXT i
PRINT "----------       ----------"

Sum1@ = 0
Sum2@ = 0
FOR i = 1 TO 10
   Sum1@ = Sum1@ + Array(i, 1)
   Sum2@ = Sum2@ + Array(i, 2)
NEXT i

PRINT Sum1@, Sum2@; " = Totale"
```

Le matrici non diventano mai molto complesse nel Basic, a differenza di quel che succede nel C, dove i nomi di matrici sono semplicemente puntatori (e i nomi di matrici bidimensionali sono solo puntatori a puntatori). In quel caso si può risparmiare tempo e memoria convertendo tutti i riferimenti a matrici in riferimenti a puntatori, ma allo stesso tempo il codice diventa estremamente difficile da leggere. Qui, invece, si è praticamente già raggiunta la massima complessità delle matrici del Basic, perciò si può passare alla tecnica successiva di organizzazione dei dati, le strutture di dati.

STRUTTURE DI DATI

Il BASIC offre la possibilità di combinare tipi di dati standard e ottenere un tipo del tutto nuovo. In effetti, in queste pagine si è usato già spesso la parola chiave TYPE per definire strutture di dati InRegs e OutRegs da usare con INTERRUPT( ) o INTERRUPTX( ) e, nel Capitolo 6 dedicato ai database ISAM, per definire record di dati. Per definire un tipo denominato Persona, si può fare così:

```
TYPE Persona
NomeProprio AS STRING * 20
Cognome AS STRING * 20
END TYPE
:
```

Questa struttura di dati non fa altro che conservare nome e cognome di una persona. Si può definire poi una variabile di questo tipo o, cosa ancor più efficace, una matrice di variabili di questo tipo:

```
TYPE Persona
NomeProprio AS STRING * 20
Cognome AS STRING * 20
END TYPE

DIM Gente(10) AS Persona
:
```

Poi si può fare riferimento alle variabili nella matrice in questo modo:

```
TYPE Persona
NomeProprio AS STRING * 20
Cognome AS STRING * 20
END TYPE

DIM Gente(10) AS Persona
```

```
Gente(1).NomeProprio = "Alfred"
Gente(1).Cognome = "Einstein"

Gente(2).NomeProprio = "Frank"
Gente(1).Cognome = "Roosevelt"

Gente(3).NomeProprio = "Francesco"
Gente(3).Cognome = "Cossiga"

PRINT Gente(1).NomeProprio
:
```

Consiglio: La parola chiave TYPE in Basic viene usata molto spesso per creare record strutturati che possono essere scritto o letti da un file ad accesso casuale (con WRITE# o INPUT#). Se a tutte le variabili di ciascun record si attribuisce un nome di tipo uniforme, è molto più facile gestirle (come si è fatto nel capitolo precedente) e si può raggiungere ciascun campo con l'operatore punto.

Le possibili strutture di dati non sono ancora finite. Se fra gli elementi di una matrice sussiste qualche collegamento, è possibile collegarli in una *lista concatenata.*

LISTE CONCATENATE

Le liste concatenate sono molto comode per organizzare dati in catene sequenziali, in particolare se si devono gestire molte catene di quel genere e si vuole sfruttare lo spazio in modo efficiente.
Nelle liste concatenate, per ogni elemento di dati esiste anche un puntatore che indica l'elemento successivo. L'ultimo puntatore della catena è un puntatore nullo con valore 0, che sta a indicare che la lista è finita (figura 7-2).
Arrivati in qualunque punto della lista, si trova l'elemento successivo facendo riferimento al puntatore dell'elemento attuale. In ogni momento si può aggiungere un nuovo elemento alla lista, semplicemente aggiornando il puntatore corrente in modo che indichi il nuovo elemento. Un esempio interessante di lista concatenata è la FAT (File Allocation Table) dei dischi: è una lista di cluster assegnati ai file per la memorizzazione. I file vengono memorizzati cluster per cluster e, per ogni cluster del disco, si trova una voce nella FAT.

Nota: *Il cluster è lo spazio minimo assegnabile su un disco; nel caso dei dischetti, i cluster sono lunghi due settori, cioè 1024 byte. Questo significa che la quantità di spazio libero sui dischi viene sempre indicata in multipli di 1024 byte.*

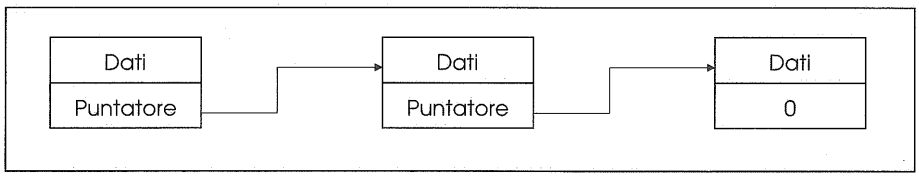

Figura 7.2

Per vedere in quali cluster è memorizzato un file, si prende il suo primo numero di cluster dai dati interni nella voce della directory: si supponga che il numero sia 2. Questo significa che la prima sezione del file è memorizzata sul disco nel cluster 2 e questo numero è anche la chiave alla FAT. Si potrà trovare il cluster *successivo* occupato dal file considerando la voce del cluster 2 nella FAT (figura 7-3).

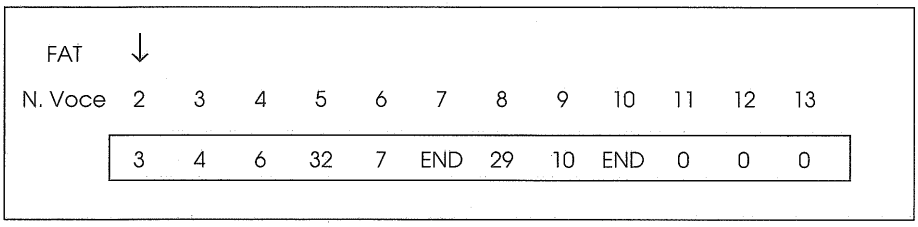

Figura 7.3

La voce di quel cluster nella FAT contiene il valore 3, che è il numero del cluster successivo occupato dal file sul disco. Per trovare il cluster dopo il 3, si va a vedere la voce di quel cluster nella FAT (figura 7-4).
Il valore che vi è contenuto è 4, perciò la sezione successiva del file si trova nel cluster 4. Per continuare, si va a vedere il numero contenuto nella voce di FAT per il cluster 4 (figura 7-5).

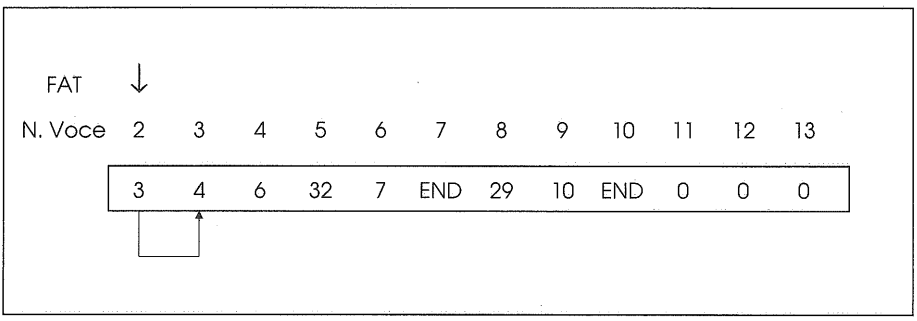

Figura 7.4

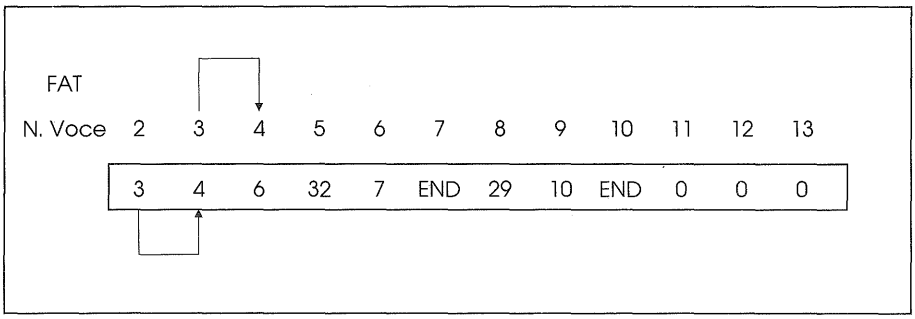

Figura 7.5

Il numero è 6, e si continua in questo modo finché non si arriva al segno di fine file (figura 7-6).

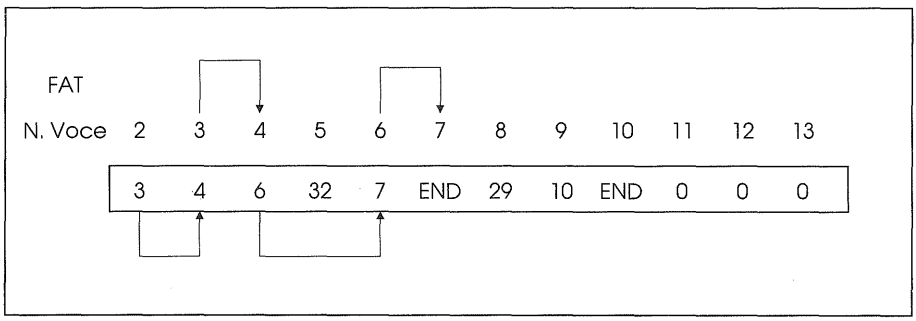

Figura 7.6

In altre parole, questo file è memorizzato nei cluster 2, 3, 4, 6 e 7. Si osservi che il cluster 5 era già stato assegnato a un altro file, che è variamente distribuito in altri cluster della FAT.

> ***Consiglio:*** Le liste concatenate si usano quando si vuole sfruttare in modo efficiente lo spazio in memoria o su disco e si deve tener traccia di numerose catene sequenziali di dati.

Quando si cancella questo file, le relative voci nella FAT possono essere sovrascritte, e quei cluster possono essere assegnati a un altro file.
Vediamo un esempio di lista concatenata in Basic. Per esempio, si può dover tener traccia di due distinti percorsi di carriera (figura 7-7):

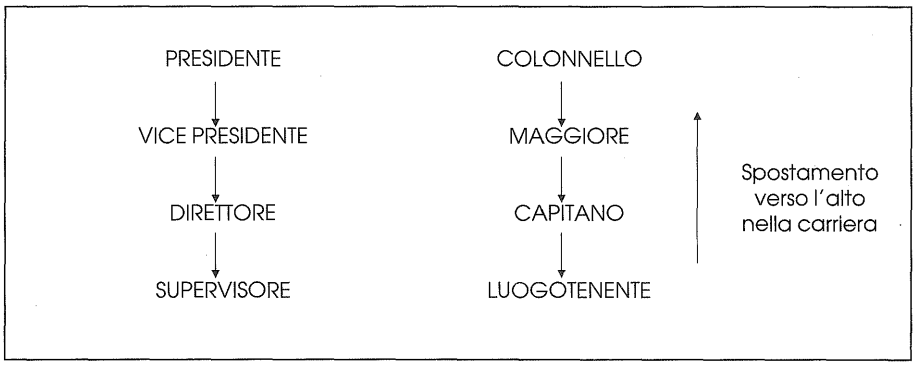

Figura 7.7

Possiamo collegare i vari livelli con una lista concatenata. Si comincia impostando una variabile di tipo Persona in questo modo:

```
REM Esempio di lista concatenata

TYPE Persona
Livello AS STRING * 20
PuntatoreASuperiore AS INTEGER
END TYPE

DIM Gente(10) AS Persona
:
```

Ora compiliamo i campi Livello: possiamo compilarli in qualsiasi ordine, poiché i puntatori li manterranno nella disposizione corretta:

```
REM Esempio di lista concatenata

TYPE Persona
Livello AS STRING * 20
PuntatoreASuperiore AS INTEGER
END TYPE

DIM Gente(10) AS Persona
Gente(1).Livello = "Supervisore"
Gente(2).Livello = "Maggiore"
Gente(3).Livello = "Direttore"
Gente(4).Livello = "Presidente"
Gente(5).Livello = "Capitano"
Gente(6).Livello = "Vice Presidente"
Gente(7).Livello = "Colonnello"
Gente(8).Livello = "Luogotenente"
:
```

Per ogni voce in Gente( ), esiste una posizione superiore. Per esempio, il superiore della voce in Gente(1), Supervisore, è in Gente(3), Direttore. Per concatenare le voci di ciascuna delle due catene, dobbiamo puntare al livello superiore compilando i puntatori Gente( ).PuntatoreASuperiore:

```
REM Esempio di lista concatenata

TYPE Persona
Livello AS STRING * 20
PuntatoreASuperiore AS INTEGER
END TYPE

DIM Gente(10) AS Persona

Gente(1).Livello = "Supervisore"
Gente(2).Livello = "Maggiore"
Gente(3).Livello = "Direttore"
Gente(4).Livello = "Presidente"
Gente(5).Livello = "Capitano"
Gente(6).Livello = "Vice Presidente"
Gente(7).Livello = "Colonnello"
Gente(8).Livello = "Luogotenente"

Gente(1).PuntatoreASuperiore = 3
Gente(2).PuntatoreASuperiore = 7
Gente(3).PuntatoreASuperiore = 6
Gente(4).PuntatoreASuperiore = 0
Gente(5).PuntatoreASuperiore = 2
Gente(6).PuntatoreASuperiore = 4
Gente(7).PuntatoreASuperiore = 0
Gente(8).PuntatoreASuperiore = 5
:
```

Ora che tutti gli elementi delle due liste sono concatenati, si può accettare un numero, 1 o 2, e percorrere di conseguenza la prima o la seconda lista, stampando i nomi dei vari Livelli (listato 7.2).

```
REM Esempio di lista concatenata

TYPE Persona
Livello AS STRING * 20
PuntatoreASuperiore AS INTEGER
END TYPE

DIM Gente(10) AS Persona

Gente(1).Livello = "Supervisore"
Gente(2).Livello = "Maggiore"
Gente(3).Livello = "Direttore"
Gente(4).Livello = "Presidente"
Gente(5).Livello = "Capitano"
Gente(6).Livello = "Vice Presidente"
Gente(7).Livello = "Colonnello"
Gente(8).Livello = "Luogotenente"

Gente(1).PuntatoreASuperiore = 3
Gente(2).PuntatoreASuperiore = 7
Gente(3).PuntatoreASuperiore = 6
Gente(4).PuntatoreASuperiore = 0
Gente(5).PuntatoreASuperiore = 2
Gente(6).PuntatoreASuperiore = 4
Gente(7).PuntatoreASuperiore = 0
Gente(8).PuntatoreASuperiore = 5

PRINT "Scegli una carriera, 1 or 2: ";
DO
    InString$ = INKEY$
LOOP WHILE InString$ = ""

PRINT

SELECT CASE InString$   'Prende il puntatore al primo record.
    CASE "1"
Indice = 1
    CASE "2"
Indice = 8
END SELECT

DO
    PRINT Gente(Indice).Livello    'Stampa i risultati.
    Indice = Gente(Indice).PuntatoreASuperiore
LOOP WHILE Indice  0
```

Listato 7.2 *Un esempio di lista concatenata.*

Per esempio, se si chiede al programma di stampare la carriera 1, si ottiene la lista:

Supervisore
Direttore
Vice Presidente
Presidente

Si badi che bisogna sapere in precedenza che la catena 1 inizia con la voce 1 e la catena 2 con la voce 8. Bisogna sempre conoscere la prima voce, che deve essere usata come chiave per la prima posizione nella lista concatenata. Dopo si può procedere sull'una o l'altra catena: così facendo, si stampa il Livello attuale e si ottiene contemporaneamente un puntatore (cioè il numero dell'indice della matrice) al Livello successivo.

BUFFER CIRCOLARI

I programmatori usano spesso un altro tipo di lista concatenata: una lista in cui l'ultimo elemento punta al primo, cosicché il tutto forma una sorta di cerchio. Questa struttura si definisce buffer circolare. Il buffer circolare meglio noto è il buffer della tastiera. Mentre una parte del sistema operativo inserisce codici di tasto nel buffer di tastiera, un'altra parte del sistema operativo li estrae. La posizione nel buffer in cui verrà inserito il successivo codice di tasto è chiamata *coda*, mentre la posizione da cui verrà letto il successivo codice di tasto è chiamata *testa*.
Quando si premono dei tasti, la coda avanza. Quando vengono letti dei codici di tasto, avanza la testa. Scrivendo sul buffer e leggendo dal buffer, testa e coda procedono circolarmente (ogni posizione può essere la posizione di testa o quella di coda). Quando il buffer è pieno, la coda si trova immediatamente dietro la testa e si attiva la segnalazione acustica di buffer pieno.

Consiglio: Si usano i buffer circolari quando una parte del programma scrive dati e un'altra parte li legge, ma a velocità diverse. Si memorizzano le posizioni di testa e di coda e, dopo aver inserito dati nel buffer, si fa avanzare la coda. Quando si estraggono dati, si fa avanzare la testa. In questo modo si può utilizzare lo stesso spazio di memoria sia per la lettura che per la scrittura.

Il problema principale delle liste concatenate, però, è che tutti gli accessi ai dati devono essere di tipo sequenziale. Per trovare l'ultima voce in una lista concatenata, per esempio, bisogna partire dalla prima e procedere a ritroso. Questo va bene per i file rintracciati attraverso la FAT (dove si deve conoscere ogni voce della FAT prima

di poter leggere tutto il file), ma è un metodo terribile se si cerca solamente un record specifico. Un metodo migliore consistere nell'impostare un *albero binario.*

ALBERI BINARI

Gli alberi binari si differenziano dalle liste concatenate in quanto, per la prima volta, si deve cominciare a ordinare i dati. Si può iniziare con una lista concatenata come indicato in figura 7.8. Poi si crea una *lista doppiamente concatenata* come indicato in figura 7.9.

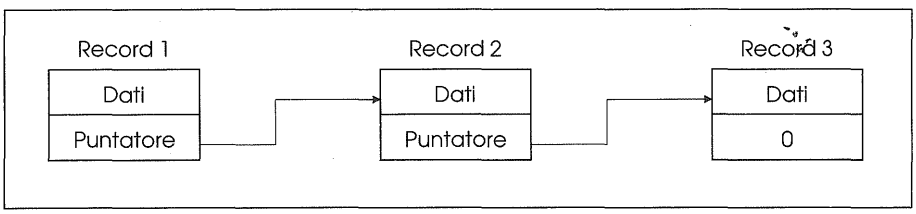

Figura 7.8

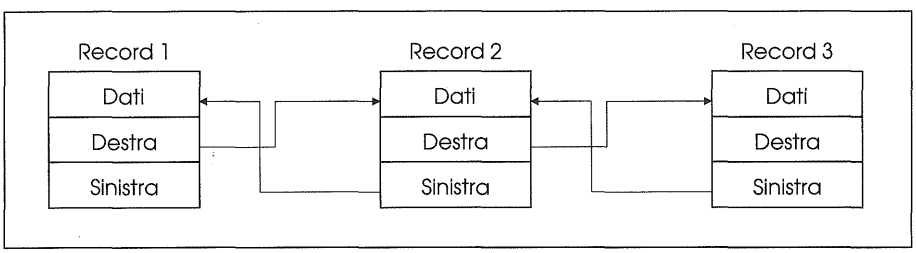

Figura 7.9

Ora ci sono due puntatori in ciascun record: uno permette di percorrere la catena verso l'alto, il secondo permette invece di percorrerla verso il basso. Le liste doppiamente concatenate hanno molti impieghi di per se stesse, ma ancora non si tratta di un albero binario. Inseriamo alcuni valori per i campi dei dati, –5, 0 e 2 (figura 7.10).

Si osservi che qui è stata costruita una gerarchia basata su valori di dati, disponendoli da sinistra a destra in ordine crescente (–5, 0, 2). Il record con i valori di dati più vicini alla mediana diventa la *radice* del nostro albero binario (figura 7.11).

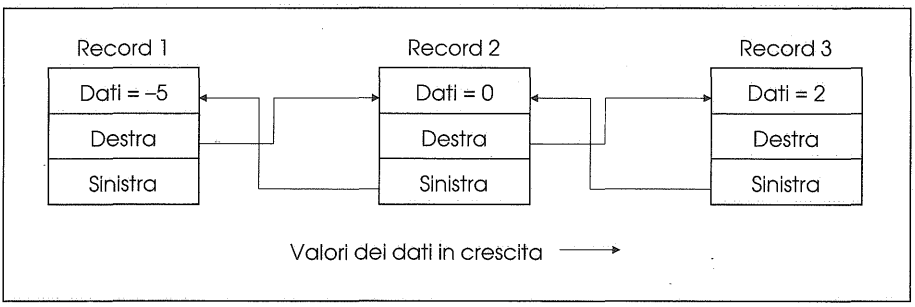

Figura 7.10

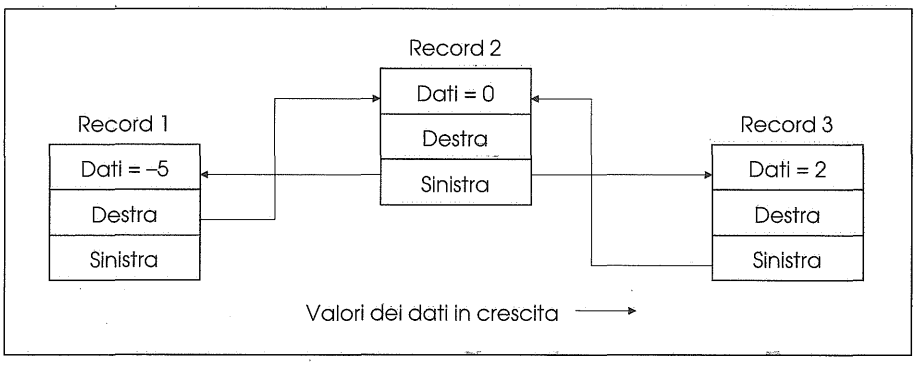

Figura 7.11

Poiché ha il dato con il valore più vicino alla mediana dei tre record, il record 2 è la radice dell'albero binario. Se si volesse trovare un record con un valore di dato, poniamo, –5, si partirebbe dalla radice, il record 2, il cui valore è 0. Poiché -5 è minore di 0, si cerca poi il record alla sinistra della radice (poiché i valori diminuiscono verso sinistra). Il record è il record 1, con un valore –5, il che significa che si è trovato il valore cercato.

Qui può sembrare che si sia guadagnato ben poco, ma immaginate di avere una lista come questa:

```
Nome = "Denise"
  Eta = 23

Nome = "Ed"
  Eta = 46

Nome = "Nick"
  Eta = 47
```

```
Nome "Dennis"
  Eta = 42

Nome = "Doug"
  Eta = 33

Nome = "Margo"
  Eta = 27

Nome = "John"
  Eta = 41

Nome = "Cheryl"
  Eta = 28
```

Supponiamo di dover coordinare questa lista e di dover cercare una persona di età specificata. Per costruire un albero binario, bisogna scegliere la persona la cui età si trova più vicina alla mediana (è Doug), per poi impostare l'albero come in figura 7.12.

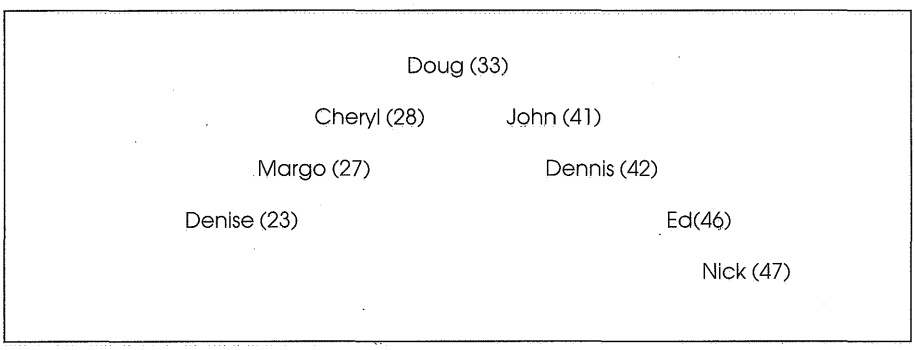

Figura 7.12

Ora si può cominciare con Doug e continuare a procedere finché non si trova la persona con l'età richiesta. Per esempio, per trovare la persona che ha 46 anni, si comincia con Doug, che ha 33 anni. Dato che 46 è maggiore di 33, si continua a spostarsi verso destra, trovando prima John, poi Dennis e infine Ed, la persona che si stava cercando.

Consiglio: Gli alberi binari sono adatti per i cosiddetti "sistemi esperti", in cui ogni nodo corrisponde a una domanda e i due rami corrispondono rispettivamente alla risposta positiva e a quella negativa.

Si badi che questo è un albero binario estremamente semplice perché, con l'eccezione di Doug, ogni nodo può diramarsi solo in una direzione. In generale, invece, i nodi possono diramarsi in ambedue le direzioni. Proviamo a codificare questo esempio.

Si comincia con il definire un nuovo tipo Persona che ha due puntatori, uno alla persona successiva in ordine di età crescente, e uno alla precedente:

```
REM Esempio di albero binario.

TYPE Persona
Nome AS STRING * 20
Eta AS INTEGER
PiuGiovaneSuccessiva AS INTEGER
PiuVecchiaSuccessiva AS INTEGER
END TYPE

DIM Gente(10) AS Persona
:
:
```

Poi si compila ciascun record:

```
REM Esempio di albero binario.

TYPE Persona
Nome AS STRING * 20
Eta AS INTEGER
PiuGiovaneSuccessiva AS INTEGER
PiuVecchiaSuccessiva AS INTEGER
END TYPE

DIM Gente(10) AS Persona

CLS
Gente(1).Nome = "Denise"
Gente(1).Eta = 23
Gente(1).PiuGiovaneSuccessiva = 0
Gente(1).PiuVecchiaSuccessiva = 6

Gente(2).Nome = "Ed"
Gente(2).Eta = 46
Gente(2).PiuGiovaneSuccessiva = 4
Gente(2).PiuVecchiaSuccessiva = 3

Gente(3).Nome = "Nick"
Gente(3).Eta = 47
Gente(3).PiuGiovaneSuccessiva = 2
Gente(3).PiuVecchiaSuccessiva = 0
```

```
Gente(4).Nome = "Dennis"
Gente(4).Eta = 42
Gente(4).PiuGiovaneSuccessiva = 7
Gente(4).PiuVecchiaSuccessiva = 2

Gente(5).Nome = "Doug"
Gente(5).Eta = 33
Gente(5).PiuGiovaneSuccessiva = 8
Gente(5).PiuVecchiaSuccessiva = 7

Gente(6).Nome = "Margo"
Gente(6).Eta = 27
Gente(6).PiuGiovaneSuccessiva = 1
Gente(6).PiuVecchiaSuccessiva = 8

Gente(7).Nome = "John"
Gente(7).Eta = 41
Gente(7).PiuGiovaneSuccessiva = 5
Gente(7).PiuVecchiaSuccessiva = 4

Gente(8).Nome = "Cheryl"
Gente(8).Eta = 28
Gente(8).PiuGiovaneSuccessiva = 6
Gente(8).PiuVecchiaSuccessiva = 5
:
:
```

Ora possiamo cercare la prima persona che ha 46 anni. Si comincia alla radice:

```
REM Esempio di albero binario.

TYPE Persona
Nome AS STRING * 20
Eta AS INTEGER
PiuGiovaneSuccessiva AS INTEGER
PiuVecchiaSuccessiva AS INTEGER
END TYPE

DIM Gente(10) AS Persona

CLS
Gente(1).Nome = "Denise"
Gente(1).Eta = 23
Gente(1).PiuGiovaneSuccessiva = 0
Gente(1).PiuVecchiaSuccessiva = 6

Gente(2).Nome = "Ed"
Gente(2).Eta = 46
Gente(2).PiuGiovaneSuccessiva = 4
```

```
Gente(2).PiuVecchiaSuccessiva = 3

Gente(3).Nome = "Nick"
Gente(3).Eta = 47
Gente(3).PiuGiovaneSuccessiva = 2
Gente(3).PiuVecchiaSuccessiva = 0

Gente(4).Nome = "Dennis"
Gente(4).Eta = 42
Gente(4).PiuGiovaneSuccessiva = 7
Gente(4).PiuVecchiaSuccessiva = 2

Gente(5).Nome = "Doug"
Gente(5).Eta = 33
Gente(5).PiuGiovaneSuccessiva = 8
Gente(5).PiuVecchiaSuccessiva = 7

Gente(6).Nome = "Margo"
Gente(6).Eta = 27
Gente(6).PiuGiovaneSuccessiva = 1
Gente(6).PiuVecchiaSuccessiva = 8

Gente(7).Nome = "John"
Gente(7).Eta = 41
Gente(7).PiuGiovaneSuccessiva = 5
Gente(7).PiuVecchiaSuccessiva = 4

Gente(8).Nome = "Cheryl"
Gente(8).Eta = 28
Gente(8).PiuGiovaneSuccessiva = 6
Gente(8).PiuVecchiaSuccessiva = 5

BinaryTreeRoot% = 5    'L'età di Doug è all'incirca la mediana

PRINT "Sto cercando una persona di 46 anni..."

CurrentRecord% = BinaryTreeRoot%
:
:
```

Poi si controlla se la persona trovata ha 46 anni:

```
REM Esempio di albero binario.

TYPE Persona
Nome AS STRING * 20
Eta AS INTEGER
PiuGiovaneSuccessiva AS INTEGER
PiuVecchiaSuccessiva AS INTEGER
```

```
END TYPE

DIM Gente(10) AS Persona

CLS
Gente(1).Nome = "Denise"
Gente(1).Eta = 23
Gente(1).PiuGiovaneSuccessiva = 0
Gente(1).PiuVecchiaSuccessiva = 6

Gente(2).Nome = "Ed"
Gente(2).Eta = 46
Gente(2).PiuGiovaneSuccessiva = 4
Gente(2).PiuVecchiaSuccessiva = 3

Gente(3).Nome = "Nick"
Gente(3).Eta = 47
Gente(3).PiuGiovaneSuccessiva = 2
Gente(3).PiuVecchiaSuccessiva = 0

Gente(4).Nome = "Dennis"
Gente(4).Eta = 42
Gente(4).PiuGiovaneSuccessiva = 7
Gente(4).PiuVecchiaSuccessiva = 2

Gente(5).Nome = "Doug"
Gente(5).Eta = 33
Gente(5).PiuGiovaneSuccessiva = 8
Gente(5).PiuVecchiaSuccessiva = 7

Gente(6).Nome = "Margo"
Gente(6).Eta = 27
Gente(6).PiuGiovaneSuccessiva = 1
Gente(6).PiuVecchiaSuccessiva = 8

Gente(7).Nome = "John"
Gente(7).Eta = 41
Gente(7).PiuGiovaneSuccessiva = 5
Gente(7).PiuVecchiaSuccessiva = 4

Gente(8).Nome = "Cheryl"
Gente(8).Eta = 28
Gente(8).PiuGiovaneSuccessiva = 6
Gente(8).PiuVecchiaSuccessiva = 5

BinaryTreeRoot% = 5    'L'età di Doug è all'incirca la mediana

PRINT "Sto cercando una persona di 46 anni..."
```

```
CurrentRecord% = BinaryTreeRoot%

DO
    IF Gente(CurrentRecord%).Eta = 46 THEN
        PRINT "La persona cercata è: ";
        PRINT "Gente(CurrentRecord%).Nome
        EXIT DO
    END IF
:
;
```

Se la risposta è negativa, si deve confrontare l'età della persona con 46: se è minore, si passa alla PiuVecchiaSuccessiva; se è maggiore, si passa alla PiuGiovaneSuccessiva. Ecco come:

```
REM Esempio di albero binario.

TYPE Persona
Nome AS STRING * 20
Eta AS INTEGER
PiuGiovaneSuccessiva AS INTEGER
PiuVecchiaSuccessiva AS INTEGER
END TYPE

DIM Gente(10) AS Persona

CLS
Gente(1).Nome = "Denise"
Gente(1).Eta = 23
Gente(1).PiuGiovaneSuccessiva = 0
Gente(1).PiuVecchiaSuccessiva = 6

Gente(2).Nome = "Ed"
Gente(2).Eta = 46
Gente(2).PiuGiovaneSuccessiva = 4
Gente(2).PiuVecchiaSuccessiva = 3

Gente(3).Nome = "Nick"
Gente(3).Eta = 47
Gente(3).PiuGiovaneSuccessiva = 2
Gente(3).PiuVecchiaSuccessiva = 0

Gente(4).Nome = "Dennis"
Gente(4).Eta = 42
Gente(4).PiuGiovaneSuccessiva = 7
Gente(4).PiuVecchiaSuccessiva = 2

Gente(5).Nome = "Doug"
Gente(5).Eta = 33
```

```
Gente(5).PiuGiovaneSuccessiva = 8
Gente(5).PiuVecchiaSuccessiva = 7

Gente(6).Nome = "Margo"
Gente(6).Eta = 27
Gente(6).PiuGiovaneSuccessiva = 1
Gente(6).PiuVecchiaSuccessiva = 8

Gente(7).Nome = "John"
Gente(7).Eta = 41
Gente(7).PiuGiovaneSuccessiva = 5
Gente(7).PiuVecchiaSuccessiva = 4

Gente(8).Nome = "Cheryl"
Gente(8).Eta = 28
Gente(8).PiuGiovaneSuccessiva = 6
Gente(8).PiuVecchiaSuccessiva = 5

BinaryTreeRoot% = 5    'L'età di Doug è all'incirca la mediana

PRINT "Sto cercando una persona di 46 anni..."

CurrentRecord% = BinaryTreeRoot%

DO
    IF Gente(CurrentRecord%).Eta = 46 THEN
PRINT "La persona cercata è: "; Gente(CurrentRecord%).Nome
EXIT DO
    END IF
    IF Gente(CurrentRecord%).Eta > 46 THEN
        CurrentRecord% =
Gente(CurrentRecord%).PiuGiovaneSuccessiva
    ELSE
        CurrentRecord% =
Gente(CurrentRecord%).PiuVecchiaSuccessiva
    END IF
LOOP WHILE CurrentRecord% <> 0
```

Ecco dunque come si effettua una ricerca in un albero binario: si continua a procedere finché si trova quel che si cercava (o si esauriscono i rami).

Con gli alberi binari, si è cominciato con l'ordinare i dati: si stabilisce, cioè, la posizione relativa di un record rispetto ai suoi due vicini. E se si volessero ordinare tutti i dati? Ordinare i dati, ovviamente, è una cosa che si fa spesso; è il passo successivo nell'organizzazione dei dati. Proprio perché è un'esigenza molto frequente, è bene esplorarla un po' dettagliatamente. Prenderemo in considerazione a questo proposito due degli algoritmi più veloci disponibili, lo shell sort e il Quicksort.

> ***Consiglio***: È difficile sapere quale sia la routine di ordinamento più adatta per un'applicazione, prima di averla vista in azione. Per questo, è bene mettere alla prova sui dati pertinenti ambedue i metodi che seguono.

SHELL SORT

Lo shell sort standard è sempre molto usato dai programmatori. Funziona in questo modo: supponiamo di avere un vettore unidimensionale che contiene questi valori:

8 7 6 5 4 3 2 1

Per disporre questi valori in ordine ascendente, si divide il vettore in due partizioni, come indicato in figura 7.13.

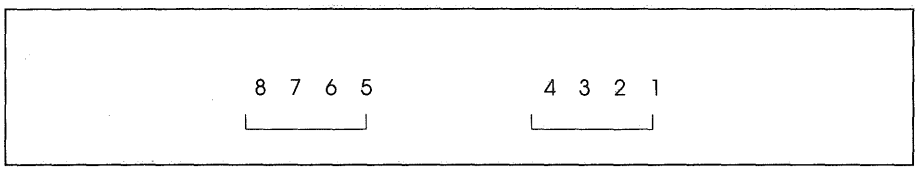

Figura 7.13

Poi si confronta il primo elemento della prima partizione con il primo elemento della seconda (figura 7.14).

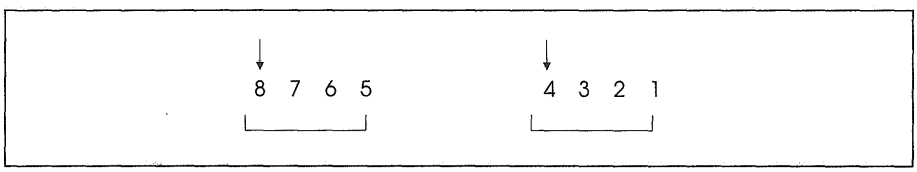

Figura 7.14

In questo caso, 8 è maggiore di 4, perciò si scambiano le posizioni dei due elementi e si passa a mettere a confronto la coppia successiva (figura 7.15).
Anche in questo caso, 7 è maggiore di 3: si scambiano di posto i due elementi e si procede (Figura 7.16).
Si scambiano di posto anche 6 e 2, poi si considera l'ultima coppia, come indicato in figura 7.17.

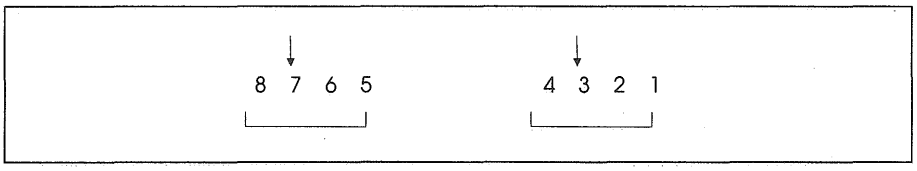

Figura 7.15

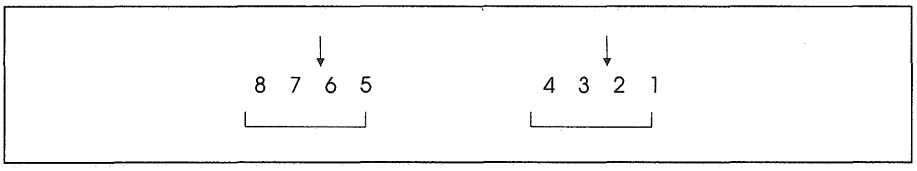

Figura 7.16

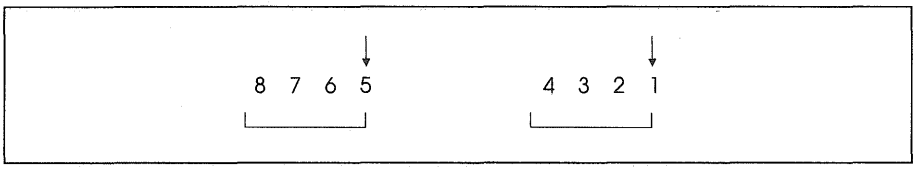

Figura 7.17

Dopo aver scambiato anche questi, si ottiene questa nuova lista:

4 3 2 1 8 7 6 5

È un po' meglio di prima, ma ancora il lavoro non è finito. Il passo successivo consiste nel dividere ciascuna partizione in due partizioni e ripetere il processo, confrontando 4 con 2 e 8 con 6 (figura 7.18).

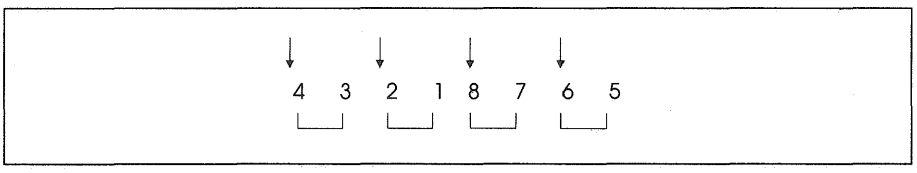

Figura 7.18

Scambiamo di posto ambedue le coppie e procediamo confrontando 3 con 1 e 7 con 5 (figura 7.19).

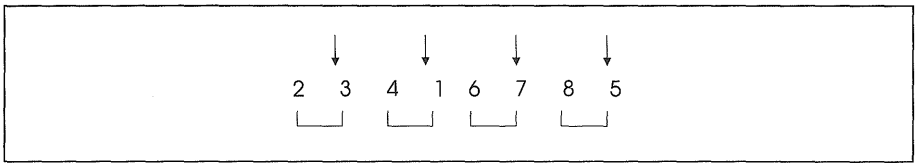

Figura 7.19

Scambiamo anche qui di posto le due coppie e otteniamo:

2 1 4 3 6 5 8 7

Siamo ancora un po' più vicini. Ora la dimensione delle partizioni si riduce a un elemento, il che significa che questa è l'ultima volta che si deve ordinare la lista. Si deve confrontare il primo e unico elemento di ciascuna partizione con il primo e unico elemento della successiva. Qui questo significa che si devono confrontare gli elementi 2, 4, 6 e 8 con gli elementi 1, 3, 5 e 7. Effettuati gli scambi, si ottiene:

1 2 3 4 5 6 7 8

E questo è il modo in cui funziona lo shell sort standard, perlomeno nel caso in cui il numero degli elementi da ordinare è pari (e in cui quindi suddividerli in partizioni equilibrate è facile). Se il numero degli elementi è dispari, la situazione è un po' più difficile. Per esempio, se si dovessero ordinare nove elementi, si dovrebbe cominbiare suddividendoli in due partizioni come in figura 7.20 (si osservi che non c'è ultimo elemento nella seconda partizione).

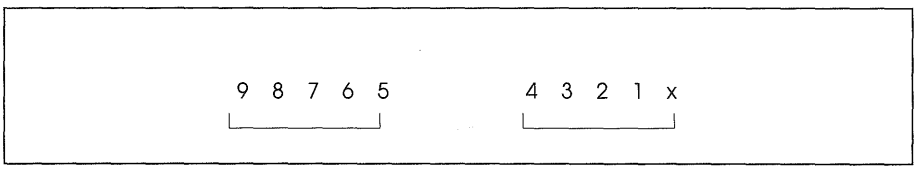

Figura 7.20

Ora si devono operare dei confronti come prima, scambiando le posizioni se necessario, finché non si deve confrontare un valore nella prima partizione con un valore nella seconda partizione che in effetti non esiste (figura 7.21).

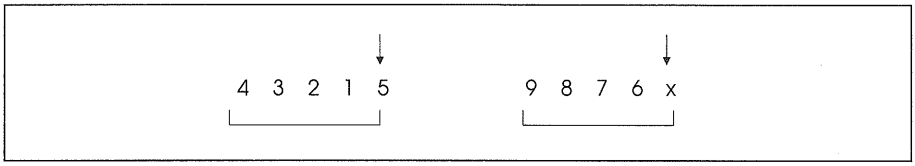

Figura 7.21

In questo caso non si esegue alcun confronto (non c'è alcun valore nella posizione indicata dalla x, che debba essere collocato in una posizione precedente nel vettore). Si continua allora al successivo livello di partizionamento del vettore. Si continua come prima, finché le partizioni contengono un solo elemento, si eseguono gli ultimi scambi di posto, e il gioco è fatto.

Ora vediamo tutto questo in Basic. Si comincia dimensionando un vettore e riempiendoli di valori (che possono essere in un ordine qualsiasi):

```
DIM Array(9) AS INTEGER

Array(1) = 9
Array(2) = 8
Array(3) = 7
Array(4) = 6
Array(5) = 5
Array(6) = 4
Array(7) = 3
Array(8) = 2
Array(9) = 1
:
:
```

Questi valori possono anche essere stampati, per poter poi essere confrontati con la lista ordinata:

```
DIM Array(9) AS INTEGER

Array(1) = 9
Array(2) = 8
Array(3) = 7
Array(4) = 6
Array(5) = 5
Array(6) = 4
Array(7) = 3
Array(8) = 2
Array(9) = 1

PRINT " i          Array(i)"
PRINT "---         --------"
```

```
FOR i = 1 TO 9
        PRINT i, Array(i)
NEXT i
PRINT
PRINT "Sorting..."
:
:
```

Ora si deve implementare lo shell sort. In questo tipo di routine di ordinamento, si cicla sulla dimensione della partizione (PartitionSize%, si veda sotto), perciò impostiamo innanzitutto questo ciclo:

```
DIM Array(9) AS INTEGER

Array(1) = 9
Array(2) = 8
Array(3) = 7
Array(4) = 6
Array(5) = 5
Array(6) = 4
Array(7) = 3
Array(8) = 2
Array(9) = 1

PRINT " i          Array(i)"
PRINT "---         --------"
FOR i = 1 TO 9
        PRINT i, Array(i)
NEXT i
PRINT
PRINT "Sorting..."

        NumItems% = UBOUND(Array, 1)
        PartitionSize% = INT((NumItems% + 1) / 2)

        DO
          :
          :
         LOOP WHILE PartitionSize%  0
                :
                :
```

La condizione che governa questo ciclo riguarda la dimensione della partizione, che si riduce alla metà a ogni passaggio nel ciclo (figura 7.22).
Per ogni dimensione di partizione, però, la lista viene suddivisa in un diverso numero di parti e bisogna impostare un ciclo su queste partizioni per poter confrontare gli elementi della partizione corrente con quelli della successiva (figura 7.23).

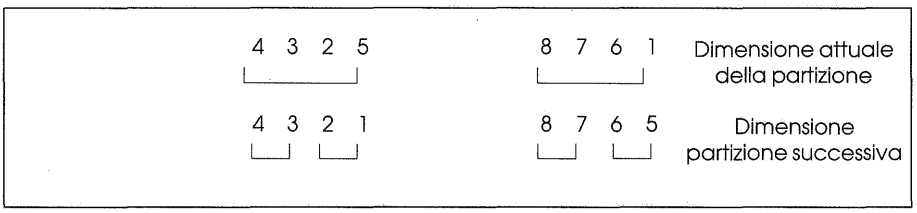

Figura 7.22

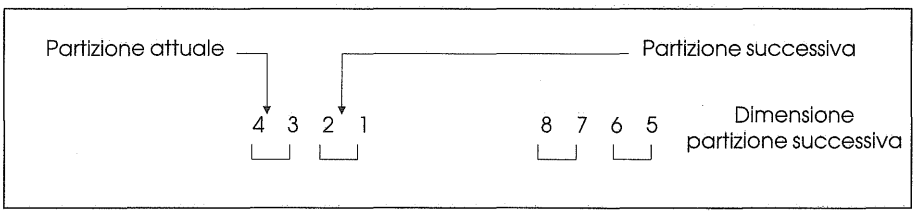

Figura 7.23

Il ciclo sulle partizioni può essere organizzato come segue (si noti che, completato il lavoro a un livello, la dimenzione delle partizioni viene dimezzata):

```
DIM Array(9) AS INTEGER

Array(1) = 9
Array(2) = 8
Array(3) = 7
Array(4) = 6
Array(5) = 5
Array(6) = 4
Array(7) = 3
Array(8) = 2
Array(9) = 1

PRINT " i           Array(i)"
PRINT "---          --------"
FOR i = 1 TO 9
        PRINT i, Array(i)
NEXT i
PRINT
PRINT "Sorting..."

        NumItems% = UBOUND(Array, 1)
        PartitionSize% = INT((NumItems% + 1) / 2)
```

```
DO
    NumPartitions% = (NumItems% + 1) / PartitionSize%
    Low% = 1
    FOR i = 1 TO NumPartitions% - 1
        :
        :
    NEXT i
    PartitionSize% = PartitionSize% \ 2
LOOP WHILE PartitionSize% > 0
:
:
```

Infine, si deve esaminare in ciclo ciascun elemento della partizione corrente, confrontandolo con l'elemento corrispondente nella partizione successiva (figura 7.24).

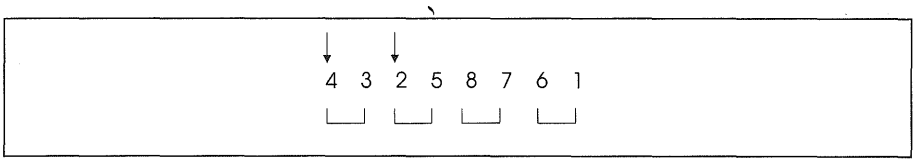

Figura 7.24

Questo è il confronto elemento per elemento. Si va da Array(Low%) ad Array(High%) nella partizione corrente, dove Low% è l'indice del primo elemento della partizione e High% è l'indice dell'ultimo, confrontando ciascun elemento con il corrispondente della partizione successiva:

```
DIM Array(9) AS INTEGER

Array(1) = 9
Array(2) = 8
Array(3) = 7
Array(4) = 6
Array(5) = 5
Array(6) = 4
Array(7) = 3
Array(8) = 2
Array(9) = 1

PRINT " i          Array(i)"
PRINT "---         --------"
FOR i = 1 TO 9
        PRINT i, Array(i)
NEXT i
PRINT
PRINT "Sorting..."
```

```
NumItems% = UBOUND(Array, 1)
PartitionSize% = INT((NumItems% + 1) / 2)

DO
    NumPartitions% = (NumItems% + 1) / PartitionSize%
    Low% = 1
    FOR i = 1 TO NumPartitions% - 1
       High% = Low% + PartitionSize% - 1
       IF High%  NumItems% - PartitionSize% THEN
           High% = NumItems% - PartitionSize%
       FOR j = Low% TO High%
          IF Array(j) > Array(j + PartitionSize%) THEN
                :
                :
          END IF
       NEXT j
       Low% = Low% + PartitionSize%
    NEXT i
    PartitionSize% = PartitionSize% \ 2
LOOP WHILE PartitionSize% > 0
:
:
```

Se risulta che l'elemento della partizione seguente è più piccolo di quello della partizione corrente, i due elementi vanno scambiati di posto, cosa che si può fare tranquillamente con l'istruzione SWAP del Basic:

```
DIM Array(9) AS INTEGER

Array(1) = 9
Array(2) = 8
Array(3) = 7
Array(4) = 6
Array(5) = 5
Array(6) = 4
Array(7) = 3
Array(8) = 2
Array(9) = 1

PRINT " i          Array(i)"
PRINT "---         --------"
FOR i = 1 TO 9
        PRINT i, Array(i)
NEXT i
PRINT
PRINT "Sorting..."

        NumItems% = UBOUND(Array, 1)
        PartitionSize% = INT((NumItems% + 1) / 2)
```

```
DO
    NumPartitions% = (NumItems% + 1) / PartitionSize%
    Low% = 1
    FOR i = 1 TO NumPartitions% - 1
       High% = Low% + PartitionSize% - 1
       IF High% > NumItems% - PartitionSize% THEN
           High% = NumItems% - PartitionSize%
       FOR j = Low% TO High%
          IF Array(j) > Array(j + PartitionSize%) THEN
            SWAP Array(j), Array(j + PartitionSize%)
          END IF
       NEXT j
       Low% = Low% + PartitionSize%
    NEXT i
    PartitionSize% = PartitionSize% \ 2
LOOP WHILE PartitionSize% > 0
:
:
```

E questo è tutto. Si cicla sulle dimensioni delle partizioni, su ciascuna partizione e su ciascun elemento della partizione corrente, scambiandolo di posto con il suo corrispondente nella partizione successiva, se necessario. Alla fine si può stampare il vettore riordinato. Il listato 7.3 mostra il programma completo.

```
DIM Array(9) AS INTEGER

Array(1) = 9
Array(2) = 8
Array(3) = 7
Array(4) = 6
Array(5) = 5
Array(6) = 4
Array(7) = 3
Array(8) = 2
Array(9) = 1

PRINT " i          Array(i)"
PRINT "---         --------"
FOR i = 1 TO 9
        PRINT i, Array(i)
NEXT i
PRINT
PRINT "Sorting..."

        NumItems% = UBOUND(Array, 1)
```

continua

Listato 7.3 *Il programma di ordinamento shell sort*

```
        PartitionSize% = INT((NumItems% + 1) / 2)

        DO
            NumPartitions% = (NumItems% + 1) / PartitionSize%
            Low% = 1
            FOR i = 1 TO NumPartitions% - 1
               High% = Low% + PartitionSize% - 1
               IF High% > NumItems% - PartitionSize% THEN
                   High% = NumItems% - PartitionSize%
               FOR j = Low% TO High%
                  IF Array(j) > Array(j + PartitionSize%) THEN
                    SWAP Array(j), Array(j + PartitionSize%)
                  END IF
               NEXT j
               Low% = Low% + PartitionSize%
            NEXT i
            PartitionSize% = PartitionSize% \ 2
        LOOP WHILE PartitionSize% > 0

PRINT
PRINT " i           Array(i)"
PRINT "---          --------"
FOR i = 1 TO 9
        PRINT i, Array(i)
NEXT i
```

Listato 7.3 *Il programma di ordinamento Shell Sort.*

Si può fare la stessa cosa anche per matrici bidimensionali. In questo caso, semplicemente si ordina la matrice in base a una delle sue colonne. Per esempio, si può adattare il programma precedente in modo che gestisca una matrice bidimensionale aggiungendo un indice di colonna (Col%) ad Array( ) (listato 7.4).

```
DIM Array(9, 4) AS INTEGER

Array(1, 1) = 9
Array(2, 1) = 8
Array(3, 1) = 7
Array(4, 1) = 6
Array(5, 1) = 5
Array(6, 1) = 4
Array(7, 1) = 3
Array(8, 1) = 2
Array(9, 1) = 1
```

continua

Listato 7.4 *Il programma di ordinamento Shell Sort bidimensionale*

```
PRINT " i           Array(i,1)"
PRINT "---          ----------"
FOR i = 1 TO 9
        PRINT i, Array(i, 1)
NEXT i
PRINT
PRINT "Sorting..."

        NumItems% = UBOUND(Array, 1)
        PartitionSize% = INT((NumItems% + 1) / 2)

        DO
            NumPartitions% = (NumItems% + 1) / PartitionSize%
            Low% = 1
            FOR i = 1 TO NumPartitions% - 1
               High% = Low% + PartitionSize% - 1
               IF High% > NumItems% - PartitionSize% THEN
                   High% = NumItems% - PartitionSize%
               FOR j = Low% TO High%
                  IF Array(j, Col%) > Array(j + PartitionSize%,
                          Col%) THEN
                    SWAP Array(j, Col%) > Array(j +
                              PartitionSize%, Col%)
                  END IF
               NEXT j
               Low% = Low% + PartitionSize%
            NEXT i
            PartitionSize% = PartitionSize% \ 2
        LOOP WHILE PartitionSize% > 0

PRINT
PRINT " i           Array(i,1)"
PRINT "---          ----------"
FOR i = 1 TO 9
        PRINT i, Array(i, 1)
NEXT i
```

Listato 7.4 *Il programma di ordinamento Shell Sort bidimensionale.*

Così siamo in grado di ordinare, secondo una colonna specifica, matrici bidimensionali. Si conclude così la rassegna dello shell sort; si potrà ora passare al QuickSort.

QUICKSORT

Il QuickSort è un altro algoritmo di ordinamento di uso frequente. La routine di ordinamento funziona in questo modo: innanzitutto si trova un valore chiave con cui confrontare gli altri valori. L'ideale sarebbe la mediana degli elementi della matrice, ma in pratica di solito si sceglie un valore a caso. Qui sceglieremo un valore dal centro della matrice.

Poi si divide la matrice in due partizioni: da una parte gli elementi minori della chiave, dall'altra quelli maggiori. Ci si sposta verso l'alto nella matrice finché si arriva al primo valore maggiore della chiave e verso il basso (partendo dalla fine) finché si trova un numero minore del valore chiave. Poi li si scambia di posto. Si continua a procedere così finché tutti i numeri della prima partizione sono inferiori alla chiave e tutti i numeri della seconda sono maggiori.

Poi si fa la stessa cosa con ciascuna partizione: si sceglie un nuovo valore chiave per ciascuna e si suddivide ciascuna partizione in due nuove partizioni. Una di queste nuove partizioni contiene i numeri inferiori alla chiave, l'altra contiene quelli maggiori. Si continua così, suddividendo progressivamente le partizioni finché ogni partizione non contiene solo due numeri, i quali vengono confrontati fra loro e scambiati di posto se necessario.

Ciascuno di questi passi è un QuickSort. Cioè, per cominciare si divide la matrice in due partizioni (valori minori e valori maggiori della chiave), poi si prende ciascuna partizione e la si suddivide in due altre partizioni in funzione di una nuova chiave e così via. Il QuickSort si presta facilmente alla ricorsione, e di solito viene codificato ricorsivamente. Anche il programma che si vedrà qui non fa eccezione.

Per chi non avesse mai trovato prima il termine *ricorsione*, va detto che si riferisce a una routine che chiama se stessa. Se un compito può essere suddiviso in una serie di livelli identici, può essere affrontato ricorsivamente. Ogni volta che la routine chiama se stessa, affronta un livello più interno; quando si raggiunge il livello finale, il controllo viene restituito al penultimo livello e così via fino al primo.

> ***Consiglio:*** La ricorsione è una delle tecniche di programmazione più potenti. Fra i suoi pregi vi sono la compattezza del codice e la facilità di messa a punto; il difetto sta nella quantità di lavoro di gestione che le chiamate successive comportano, e che produce inefficienza nell'uso della memoria e nella velocità.

Poiché questa routine è ricorsiva, si potrà definire un sottoprogramma chiamato Sort Quick( ) che verrà chiamato dal programma principale (e che poi chiamerà se stesso varie volte):

```
CALL SortQuick(Array( ), SortFrom%, SortTo%)
```

Vengono passati il nome della matrice da ordinare, l'indice da cui deve iniziare l'ordinamento (SortFrom%) e l'indice per l'ordinamento finale (SortTo%). Questo metodo di lavoro sarà utile quando si dovrà ordinare una particolare partizione della matrice.
In SortQuick( ) si affronta per primo il caso finale, cioè una partizione di due soli elementi:

```
SUB SortQuick (Array() AS INTEGER, SortFrom%, SortTo%)

        IF SortFrom% >= SortTo% THEN EXIT SUB
        IF SortFrom% + 1 = SortTo% THEN 'Caso finale della
                                        ricorsione
            IF Array(SortFrom%) > Array(SortTo%) THEN
                SWAP Array(SortFrom%), Array(SortTo%)
            END IF
        :
        :
```

In questo caso si deve semplicemente confrontare ciascun elemento con il suo vicino (l'unico altro elemento nella partizione) e li si scambia di posto se necessario. È l'unica azione nel caso finale dell'algoritmo QuickSort.
Se la dimensione della partizione è maggiore di due elementi, però, bisogna ordinare i valori da Array(SortFrom%) a Array(SortTo%), suddividendo gli elementi in due nuove partizioni in funzione di una chiave, per poi chiamare di nuovo SortQuick( ) su ciascuna partizione. Per prima cosa si sceglie una chiave, poi si divide la partizione corrente in due partizioni in base alla chiave.
Si comincia spostandosi verso l'alto dal fondo della partizione, spostando tutti i valori che si incontrano che siano maggiori della chiave:

```
SUB SortQuick (Array() AS INTEGER, SortFrom%, SortTo%)

        IF SortFrom% >= SortTo% THEN EXIT SUB
        IF SortFrom% + 1 = SortTo% THEN  'Caso finale
                                         della ricorsione
            IF Array(SortFrom%) > Array(SortTo%) THEN
                SWAP Array(SortFrom%), Array(SortTo%)
            END IF

        ELSE    'Bisogna suddividere il problema e chiamare
                 di nuovo
            AtRandom = (SortFrom% + SortTo%) \ 2
            Test = Array(AtRandom)
            SWAP Array(AtRandom), Array(SortTo%)

            DO
                REM Suddivisione in due partizioni

                FOR i = SortFrom% TO SortTo% - 1
```

```
                IF Array(i) > Test THEN EXIT FOR
            NEXT i
                :
                :
            IF i < j THEN SWAP Array(i), Array(j)
        LOOP UNTIL i >= j
            :
            :
```

Nello stesso ciclo si può anche scendere dall'alto della partizione cercando il primo valore che sia più piccolo della chiave:

```
SUB SortQuick (Array() AS INTEGER, SortFrom%, SortTo%)

        IF SortFrom% >= SortTo% THEN EXIT SUB
        IF SortFrom% + 1 = SortTo% THEN  'Caso finale
                                          della ricorsione
            IF Array(SortFrom%) > Array(SortTo%) THEN
                SWAP Array(SortFrom%), Array(SortTo%)
            END IF

        ELSE    'Bisogna suddividere il problema e chiamare
                 di nuovo
            AtRandom = (SortFrom% + SortTo%) \ 2
            Test = Array(AtRandom)
            SWAP Array(AtRandom), Array(SortTo%)

            DO
                REM Suddivisione in due partizioni

                FOR i = SortFrom% TO SortTo% - 1
                        IF Array(i) > Test THEN EXIT FOR
                NEXT i

                FOR j = SortTo% TO i + 1 STEP -1
                        IF Array(j) < Test THEN EXIT FOR
                NEXT j

                IF i < j THEN SWAP Array(i), Array(j)

            LOOP UNTIL i >= j
                :
                :
```

Si continua a procedere finché i e j non si incontrano; a quel punto sono state create le due nuove partizioni. Allora si può chiamare di nuovo SortQuick( ) per ciascuna delle partizioni risultanti (che possono anche essere di dimensioni diverse):

```
SUB SortQuick (Array() AS INTEGER, SortFrom%, SortTo%)

    IF SortFrom% >= SortTo% THEN EXIT SUB
    IF SortFrom% + 1 = SortTo% THEN  'Caso finale
                                      della ricorsione
        IF Array(SortFrom%) > Array(SortTo%) THEN
            SWAP Array(SortFrom%), Array(SortTo%)
        END IF

    ELSE    'Bisogna suddividere il problema e chiamare
             di nuovo
        AtRandom = (SortFrom% + SortTo%) \ 2
        Test = Array(AtRandom)
        SWAP Array(AtRandom), Array(SortTo%)

        DO
            REM Suddivisione in due partizioni

            FOR i = SortFrom% TO SortTo% - 1
                    IF Array(i) > Test THEN EXIT FOR
            NEXT i

            FOR j = SortTo% TO i + 1 STEP -1
                    IF Array(j) < Test THEN EXIT FOR
            NEXT j

            IF i < j THEN SWAP Array(i), Array(j)

        LOOP UNTIL i >= j

        SWAP Array(SortTo%), Array(i)

        REM Ora si lavora sulle due partizioni

        CALL SortQuick(Array(), SortFrom%, i - 1)
        CALL SortQuick(Array(), i + 1, SortTo%)
                :
                :
```

E questo è tutto: il processo di ordinamento continuerà ricorsivamente finché non viene raggiunto il caso finale di partizione di dimensione 2, e a quel punto vengono eseguiti gli ultimi scambi di posto se necessari. Il programma completo è presentato nel listato 7.5.

```
DECLARE SUB SortQuick (Array() AS INTEGER, SortFrom%, SortTo%)

DIM Array(9) AS INTEGER

Array(1) = 9
Array(2) = 8
Array(3) = 7
Array(4) = 6
Array(5) = 5
Array(6) = 4
Array(7) = 3
Array(8) = 2
Array(9) = 1

PRINT " i           Array(i)"
PRINT "---          --------"
FOR i = 1 TO 9
        PRINT i, Array(i)
NEXT i

CALL SortQuick(Array(), 1, UBOUND(Array, 1))

PRINT
PRINT "Sto ordinando..."
PRINT
PRINT " i           Array(i)"
PRINT "---          --------"
FOR i = 1 TO 9
        PRINT i, Array(i)
NEXT i


SUB SortQuick (Array() AS INTEGER, SortFrom%, SortTo%)

        IF SortFrom% >= SortTo% THEN EXIT SUB
        IF SortFrom% + 1 = SortTo% THEN  'Caso finale
                                          della ricorsione
            IF Array(SortFrom%) > Array(SortTo%) THEN
                SWAP Array(SortFrom%), Array(SortTo%)
            END IF

        ELSE    'Bisogna suddividere il problema e chiamare
                 di nuovo
            AtRandom = (SortFrom% + SortTo%) \ 2
            Test = Array(AtRandom)
            SWAP Array(AtRandom), Array(SortTo%)
```

continua

Listato 7.5 *Il programma di ordinamento QuickSort*

```
            DO

                FOR i = SortFrom% TO SortTo% - 1
                        IF Array(i) > Test THEN EXIT FOR
                NEXT i

                FOR j = SortTo% TO i + 1 STEP -1
                        IF Array(j) < Test THEN EXIT FOR
                NEXT j

                IF i < j THEN SWAP Array(i), Array(j)

            LOOP UNTIL i >= j

            SWAP Array(SortTo%), Array(i)

            CALL SortQuick(Array(), SortFrom%, i - 1)
            CALL SortQuick(Array(), i + 1, SortTo%)

        END IF

END SUB
```

Listato 7.5 *Il programma di ordinamento QuickSort.*

Come con il programma di shell sort, si può adattare il programma di QuickSort perché possa lavorare su matrici bidimensionali, mettendo le righe in ordine in base ai valori di una delle solonne (il numero di colonna è dato in Col%).

```
DECLARE SUB SortQuick (Array() AS INTEGER, SortFrom%, SortTo%,
Col%)

DIM Array(9, 4) AS INTEGER

Array(1, 1) = 9
Array(2, 1) = 8
Array(3, 1) = 7
Array(4, 1) = 6
Array(5, 1) = 5
Array(6, 1) = 4
Array(7, 1) = 3
Array(8, 1) = 2
Array(9, 1) = 1
```

continua

Listato 7.6 *Il programma di ordinamento QuickSort bidimensionale*

```
PRINT " i           Array(i,1)"
PRINT "---          ----------"
FOR i = 1 TO 9
        PRINT i, Array(i,1)
NEXT i

SUB SortQuick (Array() AS INTEGER, SortFrom%, SortTo%, Col%)

        IF SortFrom% >= SortTo% THEN EXIT SUB
        IF SortFrom% + 1 = SortTo% THEN  'Caso finale della
                                          ricorsione
            IF Array(SortFrom%, Col%) > Array(SortTo%, Col%) THEN
                SWAP Array(SortFrom%, Col%), Array(SortTo%, Col%)
            END IF

        ELSE    'Bisogna suddividere il problema e chiamare
                 di nuovo
            AtRandom = (SortFrom% + SortTo%) \ 2
            Test = Array(AtRandom, Col%)
            SWAP Array(AtRandom, Col%), Array(SortTo%, Col%)

            DO

                FOR i = SortFrom% TO SortTo% - 1
                        IF Array(i, Col%)  Test THEN EXIT FOR
                NEXT i

                FOR j = SortTo% TO i + 1 STEP -1
                        IF Array(j, Col%) > Test THEN EXIT FOR
                NEXT j

                IF i < j THEN SWAP Array(i, Col%), Array(j, Col%)

            LOOP UNTIL i >= j

            SWAP Array(SortTo%, Col%), Array(i, Col%)

            CALL SortQuick(Array(), SortFrom%, i - 1, Col%)
            CALL SortQuick(Array(), i + 1, SortTo%, Col%)

        END IF

END SUB
```

Listato 7.6 *Il programma di ordinamento QuickSort bidimensionale.*

E questo è tutto per l'ordinamento. Sia lo shell sort che il QuickSort sono molto veloci: quello che conviene usare dipende dall'applicazione. È opportuno provarli entrambi per stabilire, in ciascun caso quale risulti più veloce.

ALLA RICERCA DEI DATI

Ora che i dati sono ordinati, diventa molto più facile qualsiasi ricerca. Se i dati non sono ordinati non vi è altra scelta che controllarli uno per uno finché non se ne trova uno che corrisponde a quello che si cerca, come si può vedere nel listato 7.7.

```
REM Ricerca in una lista non ordinata

DIM Array(9) AS INTEGER

Array(1) = 9
Array(2) = 7
Array(3) = 8
Array(4) = 3
Array(5) = 5
Array(6) = 4
Array(7) = 6
Array(8) = 2
Array(9) = 1

PRINT "Sto cercando nella lista non ordinata il valore 1."

FOR i = 1 TO UBOUND(Array,)
        IF Array(i) = 1 THEN
                PRINT "Il valore 1 si trova nell'elemento";i
        END IF
NEXT i
```

Listato 7.7 *Ricerca in una lista non ordinata.*

Si continua semplicemente a esplorare la lista di valori finché non si trova quello cercato. Si può utilizzare un metodo più intelligente, invece, se si ricerca in una lista ordinata. Per esempio, se la lista ordinata contenesse i valori seguenti:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

e si cercasse un valore 10, si potrebbe partire al centro della lista, come indicato in figura 7.25.

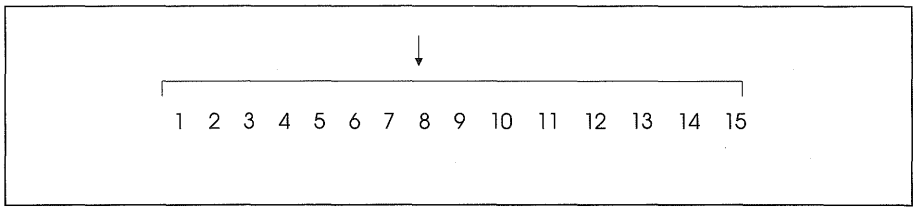

Figura 7.25

Poiché 10 è maggiore di 8, si divide in due la metà *superiore* del vettore e si va a verificare nuovamente il punto centrale (figura 7.26).

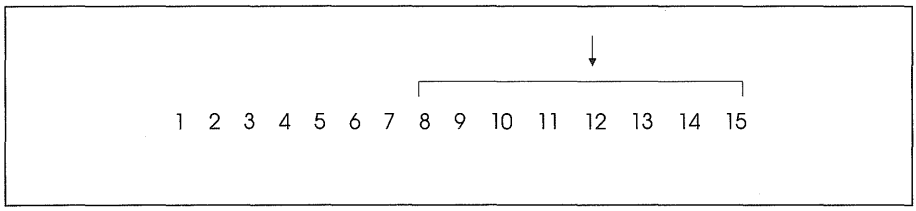

Figura 7.26

Il valore cercato è 10, che è minore di 12; perciò ci si sposta *verso il basso* e si suddivide in due metà la parte di lista rimanente (figura 7.27).

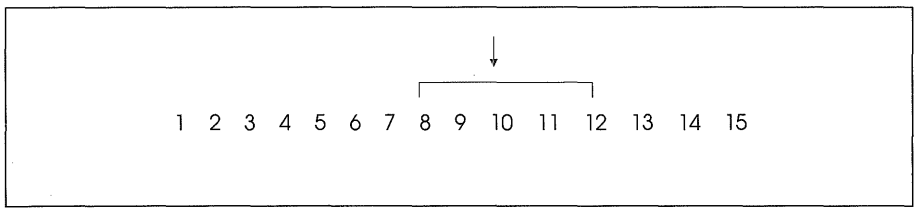

Figura 7.27

In questo modo si è arrivati proprio al numero cercato, riducendo drasticamente il numero dei valori che si sono dovuti controllare. Vediamo come il tutto si traduca in un programma.
Innanzitutto si imposta il vettore; qui si ipotizza che il vettore contenga 9 elementi e che il valore cercato sia 8:

```
REM Ricerca in una lista ordinata

DIM Array(9) AS INTEGER

Array(1) = 1
Array(2) = 2
Array(3) = 3
Array(4) = 4
Array(5) = 5
Array(6) = 6
Array(7) = 7
Array(8) = 8
Array(9) = 9

CLS
SearchValue% = 8
PRINT "Cerco nella lista ordinata il valore 8."
:
:
```

Ora si suddivide il vettore in due partizioni e si controlla il valore chiave che si trova al centro, nella posizione TestIndex%:

```
REM Ricerca in una lista ordinata

DIM Array(9) AS INTEGER

Array(1) = 1
Array(2) = 2
Array(3) = 3
Array(4) = 4
Array(5) = 5
Array(6) = 6
Array(7) = 7
Array(8) = 8
Array(9) = 9

SearchValue% = 8
PRINT "Cerco nella lista ordinata il valore 8."

Partition% = (UBOUND(Array, 1) + 1) \ 2
TestIndex% = Partition%
        :
        :
```

Poi si deve cominciare la ricerca. Si cicla sulla dimensione della partizione: se diventa 0 senza esito positivo, allora il valore cercato non si trova nella lista:

```
REM Ricerca in una lista ordinata

DIM Array(9) AS INTEGER

Array(1) = 1
Array(2) = 2
Array(3) = 3
Array(4) = 4
Array(5) = 5
Array(6) = 6
Array(7) = 7
Array(8) = 8
Array(9) = 9

SearchValue% = 8
PRINT "Cerco nella lista ordinata il valore 8."

Partition% = (UBOUND(Array, 1) + 1) \ 2
TestIndex% = Partition%

DO
    Partition% = Partition% \ 2
        :
     'Ricerca in questa partizione
        :
LOOP WHILE Partition%  0
:
:
```

Si controlla se è stato trovato il valore desiderato; se la risposta è positiva, si può uscire:

```
REM Ricerca in una lista ordinata

DIM Array(9) AS INTEGER

Array(1) = 1
Array(2) = 2
Array(3) = 3
Array(4) = 4
Array(5) = 5
Array(6) = 6
Array(7) = 7
Array(8) = 8
Array(9) = 9

SearchValue% = 8
PRINT "Cerco nella lista ordinata il valore 8."
```

```
Partition% = (UBOUND(Array, 1) + 1) \ 2
TestIndex% = Partition%

DO
    Partition% = Partition% \ 2
    IF Array(TestIndex%) = SearchValue% THEN
        PRINT "Il valore"; SearchValue%; "si trova
              nell'elemento"; TestIndex%
        EXIT DO
    END IF
        :
        :
LOOP WHILE Partition%  0
:
:
```

Se invece non è stato trovato il valore desiderato, si deve procedere alla successiva iterazione del ciclo, impostando TestIndex al valore centrale della partizione superiore o inferiore, per poi dividere quella partizione in due nuove partizioni. Se il valore cercato è maggiore del valore alla posizione corrente della lista, si deve passare alla partizione che contiene i valori maggiori:

```
REM Ricerca in una lista ordinata

DIM Array(9) AS INTEGER

Array(1) = 1
Array(2) = 2
Array(3) = 3
Array(4) = 4
Array(5) = 5
Array(6) = 6
Array(7) = 7
Array(8) = 8
Array(9) = 9

SearchValue% = 8
PRINT "Cerco nella lista ordinata il valore 8."

Partition% = (UBOUND(Array, 1) + 1) \ 2
TestIndex% = Partition%

DO
    Partition% = Partition% \ 2
    IF Array(TestIndex%) = SearchValue% THEN
        PRINT "Il valore"; SearchValue%; "si trova
              nell'elemento"; TestIndex%
        EXIT DO
```

```
    END IF
    IF Array(TestIndex%) < SearchValue% THEN
        TestIndex% = TestIndex% + Partition%
        :
        :
LOOP WHILE Partition%  0
:
:
```

Se invece il valore cercato è minore, si deve passare alla partizione inferiore (che contiene i valori minori):

```
REM Ricerca in una lista ordinata

DIM Array(9) AS INTEGER

Array(1) = 1
Array(2) = 2
Array(3) = 3
Array(4) = 4
Array(5) = 5
Array(6) = 6
Array(7) = 7
Array(8) = 8
Array(9) = 9

CLS
SearchValue% = 8
PRINT "Cerco nella lista ordinata il valore 8."

Partition% = (UBOUND(Array, 1) + 1) \ 2
TestIndex% = Partition%

DO
    Partition% = Partition% \ 2
    IF Array(TestIndex%) = SearchValue% THEN
        PRINT "Il valore"; SearchValue%; "si trova
              nell'elemento"; TestIndex%
        EXIT DO
    END IF
    IF Array(TestIndex%) < SearchValue% THEN
        TestIndex% = TestIndex% + Partition%
    ELSE
        TestIndex% = TestIndex% - Partition%
    END IF
LOOP WHILE Partition% > 0
:
:
```

Ed è quasi tutto. Si continua così finché non si trova il valore cercato, o fino a che la dimensione della partizione non diventa 0, il che significa che il valore cercato non esiste.

In caso di insuccesso, però, vi sono ancora due controlli che si devono effettuare: sul primo e sull'ultimo elemento della lista. L'algoritmo infatti richiede che tutti i numeri verificati si trovino tra due altri valori, il che è vero per tutti gli elementi del vettore meno il primo e l'ultimo. Questo significa che, se non è stato trovato il valore cercato, bisogna controllare esplicitamente anche il primo e l'ultimo elemento del vettore (listato 7.8).

```
REM Ricerca in una lista ordinata

DIM Array(9) AS INTEGER

Array(1) = 1
Array(2) = 2
Array(3) = 3
Array(4) = 4
Array(5) = 5
Array(6) = 6
Array(7) = 7
Array(8) = 8
Array(9) = 9

CLS
SearchValue% = 8
PRINT "Cerco nella lista ordinata il valore 8."

Partition% = (UBOUND(Array, 1) + 1) \ 2
TestIndex% = Partition%

DO
    Partition% = Partition% \ 2
    IF Array(TestIndex%) = SearchValue% THEN
        PRINT "Il valore"; SearchValue%; "si trova
              nell'elemento"; TestIndex%
        EXIT DO
    END IF
    IF Array(TestIndex%) < SearchValue% THEN
        TestIndex% = TestIndex% + Partition%
    ELSE
        TestIndex% = TestIndex% - Partition%
    END IF
LOOP WHILE Partition% > 0
```

continua

Listato 7.8 *Un programma di ricerca in una lista ordinata*

```
REM Può solo trovare numeri sparsi, perciò aggiungere:

    IF Array(1) = SearchValue% THEN
       PRINT "Value of"; SearchValue%; "in element 1"
    END IF

    IF Array(UBOUND(Array, 1)) = SearchValue% THEN
       PRINT "Il valore"; SearchValue%; "si trova
             nell'elemento"; UBOUND(Array, 1)
    END IF
```

Listato 7.8 *Un programma di ricerca in una lista ordinata.*

CONCLUSIONI

In questo capitolo abbiamo esaminato la maggior parte dei metodi più comuni per la gestione di dati numerici. Nel progettare un programma, la corretta organizzazione dei dati è sempre estremamente importante: come si è già detto, chi bene organizza i dati è spesso a metà dell'opera!
Presto saremo pronti per considerare la programmazione a basso livello e in linguaggio assembly; prima, però dobbiamo vedere, nel prossimo capitolo, come si effettua il debugging dei programmi.

CAPITOLO 8

DEBUGGING

Prima di affrontare il linguaggio assembly, è indispensabile passare in rassegna un aspetto importante dello sviluppo di programmi, con cui il programmatore avanzato deve avere una buona familiarità: il debugging, cioè la messa a punto. I "bachi", gli errori, capitano spesso, in particolare quando si scrivono programmi lunghi e complessi, ma i programmatori moderni hanno a disposizione, per trovarli e correggerli, strumenti davvero eccellenti e molto potenti.
In questo capitolo verranno esaminati due debugger: quello interno a QuickBASIC e il debugger indipendente CodeView.
Ciascuno dei due è adatto in determinate circostanze. Se si usa principalmente QuickBASIC, è naturale affidarsi alla finestra di Debug quando se ne ha bisogno; se si usa invece il compilatore a riga di comando, con tutta probabilità si userà CodeView per la messa a punto dei programmi. Cominciamo con il debugging di programmi sotto QuickBASIC

> ***Consiglio**:* Per la messa a punto dei programmi in linguaggio assembly di cui si parlerà nel prossimo capitolo, una buona scelta è DEBUG.COM, che viene fornito con il DOS.

DEBUGGING IN QUICKBASIC

Si supponga di ordinare alfabeticamente una decina di nomi, come questi:

John
Tim

Edward
Samuel
Frank
Todd
George
Ralph
Leonard
Thomas

Si può iniziare impostando un vettore che li contenga tutti:

```
DIM Names(10) AS STRING

Names(1) = "John"
Names(2) = "Tim"
Names(3) = "Edward"
Names(4) = "Samuel"
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"
```

Poi li si dispone in ordine alfabetico con queste istruzioni BASIC (questa parte di codice contiene due errori):

```
DIM Names(10) AS STRING
Names(1) = "John"
Names(2) = "Tim"
Names(3) = "Edward"
Names(4) = "Samuel"
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = i TO 10
        FOR j = i TO 10
                IF Names(i) > Names(j) THEN
                        Temp$ = Names(i) THEN
                        Names(j) = Names(j)
                        Names(j) = Temp$
                END IF
        NEXT j
NEXT i
:
```

Infine, si stampa il risultato, nome per nome:

```
DIM Names(10) AS STRING

Names(1)  = "John"
Names(2)  = "Tim"
Names(3)  = "Edward"
Names(4)  = "Samuel"
Names(5)  = "Frank"
Names(6)  = "Todd"
Names(7)  = "George"
Names(8)  = "Ralph"
Names(9)  = "Leonard"
Names(10)  = "Thomas"

FOR i = i TO 10
        FOR j = i TO 10
                IF Names(i) > Names(j) THEN
                        Temp$ = Names(j) THEN
                        Names(j) = Names(j)
                        Names(j) = Temp$
                END IF
        NEXT j
NEXT i
:

FOR k = 1 TO 10
        PRINT Names(k)
NEXT k
```

Purtroppo, il risultato del programma è questo:

John
Tim
Tim
Tim
Tim
Todd
Todd
Todd
Todd
Todd

Qualcosa non va, e bisogna trovare l'errore. Se si usa QuickBASIC, è facile mettere in moto il processo. La figura 8.1 mostra quale aspetto avrebbe la finestra di QuickBASIC con il programma (con il nome di ALPHA.BAS) già caricato:

```
File  Modifica  Visualizza  Ricerca  Esegui  Debug  Chiamate  Opzioni  Guida
                          ALPHA.BAS
DIM Names(10) AS STRING

Names(1) = "John"
Names(2) = "Tim"
Names(3) = "Edward"
Names(4) = "Samuel"
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = i TO 10
     FOR j = i TO 10
          IF Names(i) > Names(j) THEN
               Temp$ = Names(i)
               Names(j) = Names(j)
                          Immediata

<MAIUSC+F1=Guida> <F6=Fin.> <F2=SUB> <F5=Esegui> <F8=Passo>  |      00001:001
```

Figura 8.1

Tutto quello che si deve fare è passare alla barra del menu e selezionare l'opzione Debug. Si apre un menu come quello in figura 8.2.

```
File  Modifica  Visualizza  Ricerca  Esegui  Debug  Chiamate  Opzioni  Guida
                          ALPHA.B
DIM Names(10) AS STRING                 Osserva...
                                        Osserv. immediata...   MAIUSC+F9
Names(1) = "John"                       Punto di osservazione...
Names(2) = "Tim"                        Elimina dall'osservazione...
Names(3) = "Edward"                     Elimina ogni osservazione
Names(4) = "Samuel"
Names(5) = "Frank"                      Analizza il flusso
Names(6) = "Todd"                       Resoconto
Names(7) = "George"
Names(8) = "Ralph"                      Punto di interruzione          F9
Names(9) = "Leonard"                    Elimina ogni punto interruzione
Names(10) = "Thomas"                    Interrompi su errore
                                        Imposta istruzione successiva
FOR i = i TO 10
     FOR j = i TO 10
          IF Names(i) > Names(j) THEN
               Temp$ = Names(i)
               Names(j) = Names(j)
                          Immediata

F1=Guida | Aggiunge l'espressione specificata alla finestra di osservazione
```

Figura 8.2

Il problema ovvio di questo programma è che le voci del vettore Names( ) sono state inserite in modo scorretto. Si può osservare come venga riempito il vettore con l'opzione Add Watch. Si seleziona quest'opzione e si apre una finestra, che chiede quale sia l'espressione da tenere sotto osservazione (figura 8.3).

```
File  Modifica  Visualizza  Ricerca  Esegui  Debug  Chiamate  Opzioni  Guida
                          ALPHA.BAS
DIM Names(10) AS STRING

Names(1) = "John"
Names(2) = "Tim"
Names(3) = "Edward"
Names(4) =                    Osserva
Names(5) =
Names(6) =   Espressione da aggiungere alla finestra di osserv.:
Names(7) =
Names(8) =
Names(9) =
Names(10) =

FOR i = i T          [ OK ]      <Annulla>      <Guida>
     FOR j
          IF Names(i) > Names(j) THEN
               Temp$ = Names(i)
               Names(j) = Names(j)
                              Immediata

F1=Guida   INVIO=Agisci   ESC=Annulla   TAB=Campo succ.   FRECCIA=Voce succ.
```

Figura 8.3

I due riferimenti a Names( ) nel programma sono Names(i) e Names(j): è possibile tenere ambedue sotto osservazione. Si scrive "Names(i)" nella finestra di dialogo Add Watch per tenere sotto osservazione Names(i), poi si ripete lo stesso procedimento per Names(j). La visualizzazione cambia, come si vede in figura 8.4, e i valori correnti di Names(i) e Names(j) sono riportati nella parte alta della finestra. Ora, fermando il programma durante l'esecuzione, sarà possibile osservare quali siano i valori in Names(i) e Names(j). Per fermare l'esecuzione, si imposta un *breakpoint* che è un punto di arresto: quando viene raggiunto, l'esecuzione del programma si ferma. Per esempio, si può impostare un breakpoint spostando il cursore sullo schermo fino alla riga IF Names(i) Names(j) THEN; poi si preme F9 (oppure si seleziona Toggle Breakpoint nel menu Debug). Sullo schermo appare una linea, che sta a indicare dove è stato impostato un breakpoint (figura 8.5).

> ***Consiglio:*** Non c'è obbligo di fissare un solo breakpoint: se ne possono fissare quanti si vuole. Per esempio, se si incontrano problemi con il valore di qualche variabile, una tecnica efficace è quella di interrompere il programma in *ogni* punto in cui il suo valore viene modificato, in modo da capire che cosa succede.

```
File  Modifica  Visualizza  Ricerca  Esegui  Debug  Chiamate  Opzioni  Guida
ALPHA.BAS Names(i):
ALPHA.BAS Names(j):
                              ALPHA.BAS
DIM Names(10) AS STRING

Names(1) = "John"
Names(2) = "Tim"
Names(3) = "Edward"
Names(4) = "Samuel"
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = i TO 10
     FOR j = i TO 10
          IF Names(i) > Names(j) THEN
                              Immediata

<MAIUSC+F1=Guida> <F6=Fin.> <F2=SUB> <F5=Esegui> <F8=Passo>  |      00001:001
```

Figura 8.4

```
File  Modifica  Visualizza  Ricerca  Esegui  Debug  Chiamate  Opzioni  Guida
ALPHA.BAS Names(i):
ALPHA.BAS Names(j):
                              ALPHA.BAS
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = i TO 10
     FOR j = i TO 10
          IF Names(i) > Names(j) THEN
               Temp$ = Names(i)
               Names(j) = Names(j)
               Names(j) = Temp$
          END IF
     NEXT j
NEXT i
                              Immediata

<MAIUSC+F1=Guida> <F6=Fin.> <F2=SUB> <F5=Esegui> <F8=Passo>  |      00016:011
```

Figura 8.5

Ora si può eseguire il programma selezionando Start nel menu Run. L'esecuzione del programma continua finché non viene raggiunto il breakpoint, poi si ferma. Si possono osservare i valori di Names(i) e Names(j) visualizzati nella finestra di osservazione in alto: si scoprirà che non sono cambiati (figura 8.6).

```
File  Modifica  Visualizza  Ricerca  Esegui  Debug  Chiamate  Opzioni  Guida
ALPHA.BAS Names(i):
ALPHA.BAS Names(j):
                                ALPHA.BAS
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = i TO 10
    FOR j = i TO 10
        IF Names(i) > Names(j) THEN
            Temp$ = Names(i)
            Names(j) = Names(j)
            Names(j) = Temp$
        END IF
    NEXT j
NEXT i

                                Immediata

<MAIUSC+F1=Guida> <F5=Continua> <F9=Punto Interr> <F8=Passo> |        00016:011
```

Figura 8.6

In altre parole, la riga IF Names(i) > Names(j) THEN non confronta proprio nulla. I valori in Names(i) e Names(j) non sono validi. A questo punto del programma (cioè all'inizio) si suppone che tanto i quanto j puntino al primo elemento del vettore; sia i che j, cioè, dovrebbero essere 1. Si può verificare il valore di i semplicemente portando il cursore sulla variabile e selezionando l'opzione Instant Watch del menu Debug (figura 8.7).

Appare una finestra (visibile in figura 8.8), che visualizza il valore corrente di i.

Si vede così che i = 0, e non va bene. Bisogna cambiare questa riga nel codice e trasformarla in FOR i = 1 TO 10, perché bisogna inizializzare i prima di usarla:

```
DIM Names(10) AS STRING

Names(1) = "John"
Names(2) = "Tim"
Names(3) = "Edward"
Names(4) = "Samuel"
Names(5) = "Frank"
Names(6) = "Todd"
```

```
File  Modifica  Visualizza  Ricerca  Esegui  Debug  Chiamate  Opzioni  Guida
ALPHA.BAS Names(i):
ALPHA.BAS Names(j):
                                   ALPHA.B
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = i TO 10
     FOR j = i TO 10
          IF Names(i) > Names(j) THEN
               Temp$ = Names(i)
               Names(j) = Names(j)
               Names(j) = Temp$
          END IF
     NEXT j
NEXT i

                                   Immediata

F1=Guida | Visualizza il valore di una variabile o espressione
```

Osserva...
Osserv. immediata... MAIUSC+F9
Punto di osservazione...
Elimina dall'osservazione...
Elimina ogni osservazione

Analizza il flusso
Resoconto

Punto di interruzione F9
Elimina ogni punto interruzione
Interrompi su errore
Imposta istruzione successiva

Figura 8.7

```
File  Modifica  Visualizza  Ricerca  Esegui  Debug  Chiamate  Opzioni  Guida
ALPHA.BAS Names(i):
ALPHA.BAS Names(j):
                                   ALPHA.BAS
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "Ge
Names(8) = "Ra
Names(9) = "Le
Names(10) = "T

FOR i = i TO 1
     FOR j = i
          IF N
          END
     NEXT j
NEXT i

                                   Immediata

F1=Guida   INVIO=Agisci   ESC=Annulla   TAB=Campo succ.   FRECCIA=Voce succ.
```

Espressione
i

Valore
0

<Osserva> <Annulla> <Guida>

Figura 8.8

```
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = 1 TO 10
        FOR j = i TO 10
                IF Names(i) > Names(j) THEN
                        Temp$ = Names(i) THEN
                        Names(j) = Names(j)
                        Names(j) = Temp$
                END IF
        NEXT j
NEXT i
:

FOR k = 1 TO 10
        PRINT Names(k)
NEXT k
```

Se, però, una volta apportato questo cambiamento, si fa girare nuovamente il programma, si continua ad avere come risultato:

John
Tim
Tim
Tim
Tim
Todd
Todd
Todd
Todd
Todd

Chiaramente c'è ancora un problema. Bisogna esaminare la parte del programma in cui vengono scambiati di posti gli elementi effettivi, l'unica altra parte del programma. Si può impostare un breakpoint alla fine della parte di scambio degli elementi, come si vede in figura 8.9.
Poi si esegue il programma. L'esecuzione si ferma al breakpoint e si possono esaminare ancora Names(i) e Names(j).
Lo scambio fra gli elementi del vettore dovrebbe avvenire in questo modo: si prende il valore in Names(i) e lo si colloca in Temp$. Poi si copia l'elemento in Names(j) e lo si colloca in Names(i). Il passo finale consiste nello spostare Temp$ in Names(j). Al breakpoint, sono stati intrapresi tutti i passi meno quello finale: ora si deve spostare in Names(j) il valore che si trova in Temp$.

```
File  Modifica  Visualizza  Ricerca  Esegui  Debug  Chiamate  Opzioni  Guida
ALPHA.BAS Names(i):
ALPHA.BAS Names(j):
                              ALPHA.BAS
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = i TO 10
     FOR j = i TO 10
          IF Names(i) > Names(j) THEN
               Temp$ = Names(i)
               Names(j) = Names(j)
               Names(j) = Temp$
          END IF
     NEXT j
NEXT i
                              Immediata

<MAIUSC+F1=Guida> <F6=Fin.> <F2=SUB> <F5=Esegui> <F8=Passo>  |        00019:001
```

Figura 8.9

```
File  Modifica  Visualizza  Ricerca  Esegui  Debug  Chiamate  Opzioni  Guida
ALPHA.BAS Names(i): John
ALPHA.BAS Names(j): Edward
                              ALPHA.BAS
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = i TO 10
     FOR j = i TO 10
          IF Names(i) > Names(j) THEN
               Temp$ = Names(i)
               Names(j) = Names(j)
               Names(j) = Temp$
          END IF
     NEXT j
NEXT i
                              Immediata

<MAIUSC+F1=Guida> <F5=Continua> <F9=Punto Interr> <F8=Passo>  |        00019:016
```

Figura 8.10

In altre parole, Names(i) e Names(j) dovrebbero contenere lo stesso valore, ma così non è. In Names(i) si trova John, mentre in Names(j) c'è Edward. Qualcosa non va. Se si ritorna ad osservare attentamente il codice, si noterà questa riga:

```
DIM Names(10) AS STRING

Names(1)  = "John"
Names(2)  = "Tim"
Names(3)  = "Edward"
Names(4)  = "Samuel"
Names(5)  = "Frank"
Names(6)  = "Todd"
Names(7)  = "George"
Names(8)  = "Ralph"
Names(9)  = "Leonard"
Names(10)  = "Thomas"

FOR i = 1 TO 10
        FOR j = i TO 10
                IF Names(i) > Names(j) THEN
                        Temp$ = Names(i) THEN
                        Names(j) = Names(j)
                        Names(j) = Temp$
                END IF
        NEXT j
NEXT i
:

FOR k = 1 TO 10
        PRINT Names(k)
NEXT k
```

È evidente che questa riga dovrebbe essere invece Names(i) = names(j). Si apporta la modifica, e si ottiene il programma messo a punto (listato 8.1).

```
DIM Names(10) AS STRING
Names(1)  = "John"
Names(2)  = "Tim"
Names(3)  = "Edward"
Names(4)  = "Samuel"
Names(5)  = "Frank"
Names(6)  = "Todd"
Names(7)  = "George"
Names(8)  = "Ralph"
Names(9)  = "Leonard"
Names(10)  = "Thomas"
```

continua

Listato 8.1. *Il programma di ordinamento alfabetico dopo il debugging*

```
FOR i = 1 TO 10
        FOR j = i TO 10
                IF Names(i) > Names(j) THEN
                        Temp$ = Names(i) THEN
                        Names(i) = Names(j)
                        Names(j) = Temp$
                END IF
        NEXT j
NEXT i
:

FOR k = 1 TO 10
        PRINT Names(k)
NEXT k
```

Listato 8.1. *Il programma di ordinamento alfabetico dopo il debugging.*

E questo è il risultato finale, quando lo si manda in esecuzione:

Edward
Frank
George
John
Leonard
Ralph
Samuel
Thomas
Tim
Todd

Il programma è stato messo a punto. Ora possiamo vedere come avviene lo stesso processo in CodeView.

CODEVIEW

Consiglio: CodeView consente il debugging per tutti i linguaggi Microsoft, non solamente per il BASIC. Questa caratteristica può risultare molto comoda se si interfaccia il BASIC con il linguaggio assembly, come si farà nel capitolo successivo.

Si parte anche questa volta con il programma per mettere in ordine alfabetico la lista di nomi (con i due errori):

```
DIM Names(10) AS STRING
Names(1) = "John"
Names(2) = "Tim"
Names(3) = "Edward"
Names(4) = "Samuel"
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = i TO 10
        FOR j = i TO 10
                IF Names(i) > Names(j) THEN
                        Temp$ = Names(i) THEN
                        Names(j) = Names(j)
                        Names(j) = Temp$
                END IF
        NEXT j
NEXT i

FOR k = 1 TO 10
        PRINT Names(k)
NEXT k
```

Non si può usare con CodeView l'ambiente QB (o QBX). Bisogna invece compilare il programma, ALPHA.BAS, con BC.EXE, in questo modo:

```
D:> BC /Zi ALPHA;

Microsoft (R) QuickBASIC Compiler Version 4.50
(C) Copyright Microsoft Corporation 1982-1988.
All rights reserved.
Simultaneously published in the U.S. and Canada.

43997 Bytes Available
42104 Bytes Free

    0 Warning Error(s)
    0 Severe  Error(s)
```

Lo switch /Zi è un passo necessario nella preparazione per CodeView: dice al compilatore di conservare il numero di riga e le informazioni sui simboli di cui CodeView avrà bisogno.
Poi si effettua il link, impostando per CodeView con lo switch /Co:

```
D:> LINK /Co ALPHA;

Microsoft (R) Overlay Linker  Version 3.69
Copyright (C) Microsoft Corp 1983-1988.  All rights reserved.
```

Ed ecco fatto; si è prodotto ALPHA.EXE e si è pronti per eseguire il programma. Si ottiene però, come prima, il risultato:

John
Tim
Tim
Tim
Tim
Todd
Todd
Todd
Todd
Todd

Ovviamente c'è un problema. Si può mettere a punto ALPHA.EXE con CodeView battendo CV ALPHA. Appare la schermata di CodeView, come si può vedere in figura 8.11.

> ***Consiglio:*** CodeView può essere avviato anche in modo sequenziale (senza finestre) se si usa lo switch IT. In questa modalità, si può ridirigere il suo output a un file, il che fornisce una registrazione completa delle sessioni di messa a punto, a futura memoria.

CodeView è il debugger principe per il PC: le sue capacità sono pressoché illimitate, ma in questo capitolo riusciremo solo a scalfirne la superficie. Si può partire come con QuickBASIC, tenendo sotto osservazione Names(i) e Names(j). Si apre, per questo, il menu Watch e si seleziona Add Watch (figura 8.12).
Si apre la finestra mostrata in figura 8.13, molto simile alla finestra del QuickBASIC. Si scrive Names(i), poi si ripete il procedimento per Names(j).
La finestra si suddivide in due e appare una finestra di osservazione, come si può vedere in figura 8.14.

> ***Consiglio:*** Le finestre di CodeView possono scorrere indipendentemente l'una dall'altra; se un'espressione esce dallo schermo, si possono sempre utilizzare le barre di scorrimento per visualizzarla completamente.

Si può inserire un breakpoint, come si è fatto con il QuickBASIC, alla linea "IF Names(i) > Names(j)" THEN con il tasto funzione F9 (oppure con il comando Set

```
File  Edit  View  Search  Run  Watch  Options  Calls                    Help
──────────────────────────────────local──────────────────────────────────
──────────────────────source1 CS:IP ALPHA.BAS (ACTIVE)──────────────────
1:      DIM Names(10) AS STRING
2:
3:      Names(1) = "John"
4:      Names(2) = "Tim"
5:      Names(3) = "Edward"
6:      Names(4) = "Samuel"
7:      Names(5) = "Frank"
8:      Names(6) = "Todd"
9:      Names(7) = "George"
10:     Names(8) = "Ralph"
11:     Names(9) = "Leonard"
12:     Names(10) = "Thomas"
13:
──────────────────────────────command──────────────────────────────────
>
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>
```

Figura 8.11

```
File  Edit  View  Search  Run  Watch  Options  Calls                    Help
                              Add Watch...          Ctrl+W
                              Delete Watch...       Ctrl+U
                              Set Breakpoint...        F9
                              Edit Breakpoints...
                              Quick Watch...     Shift+F9
                     source1
10:     Names(8) = "Ralph"
11:     Names(9) = "Leonard"
12:     Names(10) = "Thomas"
13:
14:     FOR i = i TO 10
15:             FOR j = i TO 10
16:                     IF Names(i) > Names(j) THEN
17:                             Temp$ = Names(i)
18:                             Names(j) = Names(j)
19:                             Names(j) = Temp$
20:                     END IF
──────────────────────────────command──────────────────────────────────
>
>
>
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display> <ESC=Cancel>
```

Figura 8.12

```
File  Edit  View  Search  Run  Watch  Options  Calls                Help
local
source1 CS:IP ALPHA.BAS (ACTIVE)
10:     Names(8) = "Ralph"
11:     Name
12:     Name
13:
14:     FOR
15:
16:
17:
18:
19:                     Names(j) = Temp$
20:                 END IF
command
>
>
>
<F1=Help> <Enter> <ESC=Cancel> <TAB=Next Field>
```

Add Watch

Expression: [Names(i)]

< OK > <Cancel> < Help >

Figura 8.13

```
File  Edit  View  Search  Run  Watch  Options  Calls                Help
local                                watch
                                     Names(i) <Watch Expression Not In Con
                                     Names(j) <Watch Expression Not In Con
source1 CS:IP ALPHA.BAS (ACTIVE)
13:
14:     FOR i = i TO 10
15:             FOR j = i TO 10
16:                     IF Names(i) > Names(j) THEN
17:                             Temp$ = Names(i)
18:                             Names(j) = Names(j)
19:                             Names(j) = Temp$
20:                     END IF
21:             NEXT j
22:     NEXT i
23:
command
>
>
>
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>
```

Figura 8.14

Breakpoint nella finestra Watch). CodeView è un debugger guidato da comandi: per eseguire il programma fino al punto di interruzione, basta battere g, che sta per go, ovvero "vai" (figura 8.15).

```
File  Edit  View  Search  Run  Watch  Options  Calls                    Help
local                                  watch
                                       Names(i) <Watch Expression Not In Con
                                       Names(j) <Watch Expression Not In Con
source1 CS:IP ALPHA.BAS (ACTIVE)
10:     Names(8) = "Ralph"
11:     Names(9) = "Leonard"
12:     Names(10) = "Thomas"
13:
14:     FOR i = i TO 10
15:             FOR j = i TO 10
16:                     IF Names(i) > Names(j) THEN
17:                             Temp$ = Names(i)
18:                             Names(j) = Names(j)
19:                             Names(j) = Temp$
20:                     END IF
21:             NEXT j
command
>g
>
>
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>
```

Figura 8.15

L'esecuzione continua finché non raggiunge il breakpoint, poi si può esaminare la finestra di osservazione. Come si vede in figura 8.16, Names(i) e Names(j) sono ancora stringhe nulle, mentre dovrebbero essere già state assegnate delle stringhe di caratteri. Si può controllare il valore degli indici i e j con il comando ?. In questo caso, il comando "?i" dice a CodeView di visualizzare il valore corrente di i (figura 8.17), e si vede che è 0. Anche in questo caso si vede subito che questa linea del programma provoca problemi:

```
DIM Names(10) AS STRING
Names(1) = "John"
Names(2) = "Tim"
Names(3) = "Edward"
Names(4) = "Samuel"
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"
```

```
File  Edit  View  Search  Run  Watch  Options  Calls                    Help
──────────────────local──────────────────  ──────────────────watch──────────────
3266:0DF2 SINGLE          I! = 0           Names(i) = ""
3266:0DF6 SINGLE          J! = 0           Names(j) = ""
3266:0DFE SINGLE          K! = 0
──────────────────source1 CS:IP ALPHA.BAS (ACTIVE)──────────────────────────
12:     Names(10) = "Thomas"
13:
14:     FOR i = i TO 10
15:             FOR j = i TO 10
16:                     IF Names(i) > Names(j) THEN
17:                             Temp$ = Names(i)
18:                             Names(j) = Names(j)
19:                             Names(j) = Temp$
20:                     END IF
21:             NEXT j
22:     NEXT i
23:
──────────────────────────────────command──────────────────────────────────
Break at: "ALPHA.BAS"!.16
>
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>
```

Figura 8.16

```
File  Edit  View  Search  Run  Watch  Options  Calls                    Help
──────────────────local──────────────────  ──────────────────watch──────────────
3266:0DF2 SINGLE          I! = 0           Names(i) = ""
3266:0DF6 SINGLE          J! = 0           Names(j) = ""
3266:0DFE SINGLE          K! = 0
──────────────────source1 CS:IP ALPHA.BAS (ACTIVE)──────────────────────────
12:     Names(10) = "Thomas"
13:
14:     FOR i = i TO 10
15:             FOR j = i TO 10
16:                     IF Names(i) > Names(j) THEN
17:                             Temp$ = Names(i)
18:                             Names(j) = Names(j)
19:                             Names(j) = Temp$
20:                     END IF
21:             NEXT j
22:     NEXT i
23:
──────────────────────────────────command──────────────────────────────────
Break at: "ALPHA.BAS"!.16
>?i
0
>
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>
```

Figura 8.17

```
FOR i = i TO 10
        FOR j = i TO 10
                IF Names(i) > Names(j) THEN
                        Temp$ = Names(i) THEN
                        Names(j) = Names(j)
                        Names(j) = Temp$
                END IF
        NEXT j
NEXT i

FOR k = 1 TO 10
        PRINT Names(k)
NEXT k
```

> ***Consiglio:*** CodeView può anche valutare delle espressioni BASIC. Per esempio, ?ASC(A"A") + 5 darà il risultato 70. Questa capacità offre un buon metodo per controllare il codice, quando non si è sicuri del risultato di talune istruzioni, e lo si può fare senza lasciare CodeView.

Si può cambiarla in FOR i = 1 TO 10. Sfortunatamente, il programma parzialmente messo a punto continua a dare lo stesso risultato:

John
Tim
Tim
Tim
Tim
Todd
Todd
Todd
Todd
Todd

Si può tornare allora a CodeView. Questa volta si inserisce un breakpoint più avanti, alla fine della selezione che scambia di posto gli elementi nel vettore (figura 8.18). Ancora una volta si batte g (per go) e il programma viene eseguito fino al punto di interruzione. Come prima, la sezione di codice che scambia di posto gli elementi del vettore opera in tre passi. Innanzitutto sposta Names(i) in Temp$, poi sposta Names(j) in Names(i), e infine sposta Temp$ in Names(j). Al momento dell'interruzione sono stati completati i primi due passi e manca l'ultimo: questo significa che Names(i) e Names(j) ora devono contenere gli stessi valori. In realtà, Names(i) = "John" e Names(j) = "Edward" (figura 8.19).
Questo indica ancora una volta l'esistenza di un problema nel modo in cui gli elementi del vettore vengono caricati durante il processo di scambio di posto. Si

```
File  Edit  View  Search  Run  Watch  Options  Calls                    Help
local                                  watch
                                       Names(i) <Watch Expression Not In Con
                                       Names(j) <Watch Expression Not In Con
source1 CS:IP ALPHA.BAS (ACTIVE)
10:     Names(8) = "Ralph"
11:     Names(9) = "Leonard"
12:     Names(10) = "Thomas"
13:
14:     FOR i = 1 TO 10
15:             FOR j = i TO 10
16:                     IF Names(i) > Names(j) THEN
17:                             Temp$ = Names(i)
18:                             Names(j) = Names(j)
19:                             Names(j) = Temp$
20:                     END IF
command
>
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>
```

Figura 8.18

```
File  Edit  View  Search  Run  Watch  Options  Calls                    Help
local                                  watch
3266:0DF2 SINGLE        I! = 1         Names(i) = "John"
3266:0DF6 SINGLE        J! = 3         Names(j) = "Edward"
3266:0DFE SINGLE        K! = 0
source1 CS:IP ALPHA.BAS (ACTIVE)
10:     Names(8) = "Ralph"
11:     Names(9) = "Leonard"
12:     Names(10) = "Thomas"
13:
14:     FOR i = 1 TO 10
15:             FOR j = i TO 10
16:                     IF Names(i) > Names(j) THEN
17:                             Temp$ = Names(i)
18:                             Names(j) = Names(j)
19:                             Names(j) = Temp$
20:                     END IF
21:             NEXT j
22:     NEXT i
command
>g
Break at: "ALPHA.BAS"!.19
>
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>
```

Figura 8.19

controlla il codice e si scopre che la linea immediatamente precedente il breakpoint contiene un errore:

```
DIM Names(10) AS STRING
Names(1) = "John"
Names(2) = "Tim"
Names(3) = "Edward"
Names(4) = "Samuel"
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = 1 TO 10
        FOR j = i TO 10
                IF Names(i) > Names(j) THEN
                        Temp$ = Names(i) THEN
                        Names(j) = Names(j)
                        Names(j) = Temp$
                END IF
        NEXT j
NEXT i

FOR k = 1 TO 10
        PRINT Names(k)
NEXT k
```

La si corregge in Names(i) = Names(j) e ottenendo il programma corretto (listato 8.2).

```
DIM Names(10) AS STRING
Names(1) = "John"
Names(2) = "Tim"
Names(3) = "Edward"
Names(4) = "Samuel"
Names(5) = "Frank"
Names(6) = "Todd"
Names(7) = "George"
Names(8) = "Ralph"
Names(9) = "Leonard"
Names(10) = "Thomas"

FOR i = 1 TO 10
        FOR j = i TO 10
                IF Names(i) > Names(j) THEN
```

continua

Listato 8.2. *Il programma di ordinamento alfabetico dopo il debugging*

```
                        Temp$ = Names(i) THEN
                        Names(i) = Names(j)
                        Names(j) = Temp$
                END IF
        NEXT j
NEXT i

FOR k = 1 TO 10
        PRINT Names(k)
NEXT k
```

Listato 8.2. *Il programma di ordinamento alfabetico dopo il debugging.*

Il risultato ora è corretto:

Edward
Frank
George
John
Leonard
Ralph
Samuel
Thomas
Tim
Todd

Il programma è stato messo a punto. CodeView è un debugger potente e offre al programmatore moltissimi strumenti. Val la pena di notare che, oltre ai breakpoint, CodeView ha anche *tracepoint* e *watchpoint*.
Un tracepoint è un breakpoint variabile che si attiva quando cambia un determinato valore, che può essere il valore di una variabile o di qualche espressione. Per esempio, si può attivare il tracepoint quando cambia il valore di una determinata variabile.
Un watchpoint è un breakpoint variabile che si attiva quando diventa vera (cioè diversa da zero) una determinata espressione. In altre parole, si può assegnare a CodeView un'espressione BASIC da controllare durante l'esecuzione del programma: quando quell'espressione diventa vera, viene attivato il watchpoint. Qui abbiamo solo scalfito la superficie, ma se si generano errori gravi, CodeView è in grado di fornire un'ottima assistenza nella maggior parte dei casi.

Nota: *CodeView non aiuta nel debugging di file residenti in memoria, e non può conservare informazioni simboliche in programmi compilati come file .COM.*

Termina così la nostra discussione del debugging. Per ulteriori informazioni, si consultino i manuali del QuickBASIC o di CodeView. Ora, invece, è ora di passare al linguaggio assembly per aggiungere davvero velocità e potenza ai programmi in BASIC. Come primo passo, si vedrà che cosa può offrire il sistema operativo.

CAPITOLO 9

ANALISI DEL BIOS E DEL DOS

Uno dei metodi più potenti per arricchire il BASIC consiste nell'uso delle risorse disponibili attraverso il BIOS (il Basic Input/Output System che si trova già nelle ROM del calcolatore) e il DOS: il BASIC consente di raggiungere direttamente le risorse di basso livello della macchina mediante le sue routine INTERRUPT( ) e INTERRUPTX( ).

> ***Consiglio:*** Spesso, ciò che fa il BASIC può essere fatto molto più rapidamente interfacciandosi con il DOS e il BIOS: l'esempio tipico è la manipolazione veloce dei file.

In questo capitolo passeremo in rassegna il BIOS e il DOS ed esamineremo tutti i servizi che questi forniscono (e che si trovano elencati nell'appendice alla fine del libro). Questi servizi sono interni al sistema operativo, il che significa che sono sempre disponibili a chi programma in BASIC. Non limiteremo la nostra esplorazione a una semplice occhiata, ma costruiremo anche dei sottoprogrammi e delle funzioni che ci permetteranno di sfruttare questa potenza aggiuntiva.
Per esempio, scriveremo routine che possono andare alla ricerca di file che soddisfano una determinata specifica (con la possibilità di usare carattery jolly), una routine che può stabilire la quantità di spazio libero che resta su un dato disco, una routine che può dire quali dispositivi siano installati nel calcolatore, e altre ancora. Per sviluppare tutto questo, iniziamo con gli interrupt del BIOS.

GLI INTERRUPT DEL BIOS

Agli interrupt del BIOS sono assegnati numeri che vanno da 0 a &H1F. Ciascuno di questi interrupt corrisponde a codice in linguaggio assembly già residente in memoria e raggiungibile attraverso le istruzioni INTERRUPT( ) o INTERRUPTX( ). Gli interrupt di basso livello del BIOS (fino a &H10) non vengono utilizzati spesso dai programmatori, fatta eccezione per l'interrupt 9, che legge i tasti dalla tastiera. Ogni volta che si preme un tasto, viene eseguito un interrupt 9: è possibile *intercettare* questo interrupt e gestire le battute direttamente dal proprio programma, il che è alla base di molti programmi di tipo TSR. Ecco quello che fanno gli interrupt da 0 a &HF:

Interrupt 0	Divisione per 0 (generato se un programma divide per 0)
Interrupt 1	Passo singolo (usato dai debugger)
Interrupt 2	Interrupt hardware non mascherabile (NMI)
Interrupt 3	Breakpoint (utilizzato dai debugger)
Interrupt 4	Overflow
Interrupt 5	Stampa video
Interrupt 6 e 7	Riservati
Interrupt 8	Ora
Interrupt 9	Tastiera
Interrupt &HA	Riservato
Interrupt &HB-&HF	Routine di interrupt di fine del BIOS, gestore dell'interrupt di reset

In particolare va notato l'interrupt 5, che viene chiamato quando si preme il tasto STAMP: se si esegue un interrupt 5 con INTERRUPT( ), si produce una stampa del video. Il primo interrupt rilevante del BIOS è &H10, che gestisce lo schermo al livello più basso. Qui sono forniti questi servizi:

Interrupt &H10	Servizio 0	Imposta il modo schermo
Interrupt &H10	Servizio 1	Imposta il tipo di cursore
Interrupt &H10	Servizio 2	Imposta la posizione del cursore
Interrupt &H10	Servizio 3	Trova la posizione del cursore
Interrupt &H10	Servizio 4	Legge la posizione della penna ottica
Interrupt &H10	Servizio 5	Imposta la pagina video attiva
Interrupt &H10	Servizio 6	Fa scorrere verso l'alto la pagina attiva
Interrupt &H10	Servizio 7	Fa scorrere verso il basso la pagina attiva
Interrupt &H10	Servizio 8	Legge attributo/carattere alla posizione del cursore
Interrupt &H10	Servizio 9	Scrive attributo/carattere alla posizione del cursore
Interrupt &H10	Servizio &HA	Scrive un carattere solo alla posizione del cursore

Interrupt &H10	Servizio &HB	Imposta la tavola dei colori
Interrupt &H10	Servizio &HC	Scrive un punto (imposta un pixel)
Interrupt &H10	Servizio &HD	Legge un punto (legge il colore di un pixel)
Interrupt &H10	Servizio &HE	Scrittura sulla pagina attiva
Interrupt &H10	Servizio &HF	Restituisce lo stato del video
Interrupt &H10	Servizio &H10	Imposta i registri della tavola
Interrupt &H10	Servizio &H10	Funzione 0: imposta un singolo registro della tavola
Interrupt &H10	Servizio &H10	Funzione 1: imposta il registro di overscan (bordo)
Interrupt &H10	Servizio &H10	Funzione 2: imposta tutti i registri della tavola
Interrupt &H10	Servizio &H10	Funzione 7: legge un singolo registro della tavola
Interrupt &H10	Servizio &H10	Funzione 8: legge il registro di overscan (bordo)
Interrupt &H10	Servizio &H10	Funzione &H10: imposta un registro DAC
Interrupt &H10	Servizio &H10	Funzione &H12: imposta i registri DAC
Interrupt &H10	Servizio &H10	Funzione &H13: seleziona la modalità pagina a colori
Interrupt &H10	Servizio &H11	Generatore di caratteri
Interrupt &H10	Servizio &H12	Selezione alternativa

Per selezionare un servizio da un interrupt, si carica il numero del servizio nel registro AH. Alcuni di questi servizi sono piuttosto complessi e non avrebbe senso dedicarvi troppo tempo in una semplice rassegna delle risorse di sistema. Tuttavia, spesso è utile sapere quale sia la modalità del video corrente, perciò scriveremo una piccola funzione per trovare la modalità video BIOS corrente.

IL MODO SCHERMO DEL BIOS

Se si sa quale sia il modo BIOS corrente, si sa quante sono le righe e le colonne disponibili sullo schermo (o qual è il numero dei pixel disponibili) e il numero dei colori visualizzabili contemporaneamente.

Ecco un elenco dei modi BIOS, con il relativo significato per quanto riguarda le possibilità del video:

Mod. BIOS	**Righe visualizz.**	**Numero colori**	**Scheda grafica**	**N. max pag. ammesse**
0	40x25	testo mono	CGA, EGA, VGA	8
1	40x25	testo colore	CGA, EGA, VGA	8
2	80x25	testo mono	CGA, EGA, VGA	4 (CGA), 8 (EGA, VGA)

Mod. BIOS	Righe visualizz.	Numero colori	Scheda grafica	N. max pag. ammesse
3	80x25	testo colore	CGA, EGA, VGA	4 (CGA), 8 (EGA, VGA)
4	320x200	4	CGA, EGA, VGA	1
5	320x200	mono	CGA, EGA, VGA	1
6	600x200	2 (on o off)	CGA, EGA, VGA	1
7	80x25	mono	MDA, EGA, VGA	1 (MDA), 8 (EGA, VGA)
8	160x200	16	PCjr	1
9	320x200	16	PCjr	1
A	640x200	1	PCjr	1
B	Riservato per uso futuro			
C	Riservato per uso futuro			
D	320x200	16	EGA, VGA	8
E	640x200	16	EGA, VGA	4
F	640x350	mono	EGA, VGA	2
10H	640x350	16	EGA, VGA	2
11H	640x480	2	VGA	1
12H	640x480	16	VGA	1
13H	320x200	256	VGA	1

Per conoscere il modo video BIOS, consultando l'appendice si vede che quello che serve è il servizio &HF dell'interrupt &H10.
Si dia il nome VideoMode%( ) alla funzione da scrivere, che restituirà il modo video. Si inizia caricando in ah &HF e chiamando INTERRUPT( ):

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.ax = &HF00
CALL INTERRUPT(&H10, InRegs, OutRegs)
:
```

In appendice si può vedere che il modo video viene restituito in al, gli otto bit inferiori del registro ax. Per averlo come valore della funzione VideoMode%, si effettua l'AND di ax con &HFF:

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.ax = &HF00
CALL INTERRUPT(&H10, InRegs, OutRegs)
VideoMode% = &HFF AND OutRegs.ax
```

Ed è tutto. Il listato 9.1 presenta la funzione completa.

```
DECLARE FUNCTION VideoMode% ()

TYPE RegType
        ax      AS INTEGER
        bx      AS INTEGER
        cx      AS INTEGER
        dx      AS INTEGER
        bp      AS INTEGER
        si      AS INTEGER
        di      AS INTEGER
        flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

FUNCTION VideoMode%

   DIM InRegs AS RegType, OutRegs AS RegType
   InRegs.ax = &HF00
   CALL INTERRUPT(&H10, InRegs, OutRegs)
   VideoMode% = &HFF AND OutRegs.ax

END FUNCTION
```

Listato 9.1. *La funzione VideoMode% determina il modo video.*

Ed ecco un programmino che mette al lavoro VideoMode%( ):

```
REM Esempio di uso di VideoMode

DECLARE FUNCTION VideoMode% ( )

Mode% = VideoMode%

PRINT "Il modo video è "; Mode%
```

Quando lo si esegue, alla variabile Mode% viene attribuito il valore restituito dalla funzione VideoMode%( ); il valore viene poi visualizzato.
Il successivo interrupt BIOS è &H11, relativo alla determinazione dei dispositivi: stabilisce quale tipo di hardware sia installato nel calcolatore. Si tratta di informazioni spesso molto utili per chi programma in BASIC.

COME DETERMINARE L'HARDWARE INSTALLATO NEL COMPUTER

L'interrupt &H11 è molto potente. Per esempio, lo si può usare per vedere se sono installati un coprocessore matematico, un modem interno, una stampante oppure un mouse. L'interrupt riempie il registro AX, bit per bit, in questo modo:

bit di AX	**indica**
15, 14	Numero di stampanti installate
13	È installato un modem interno? (1 = sì)
12	Non utilizzato
11, 10, 9	Numero di schede RS-232 collegate
8	Non utilizzato
7, 6	Numero di drive per floppy disk: 00 = 1 drive, 01 = 2 drive
5, 4	Tipo di video: 01 = 40x25 a colori, 10 = 80x25 a colori, 11 = 80x25 monocromatico
3	Non utilizzato
2	È installato un mouse? (1 = sì)
1	È installato un coprocessore matematico? (1 = sì)
0	Sono installati drive per floppy disk? (1 = sì)

Sviluppiamo una piccola funzione, GetEquipment%( ), in grado di restituire questo tipo di informazioni.
Il codice effettivo è quanto di più semplice: si fa solo uso dell'interrupt &H11 e, al ritorno della funzione, i valori dei bit del registro ax indicheranno le condizioni dell'hardware. Si può impostare GetEquipment% a quei valori e tornare:

```
DIM InRegs AS RegType, OutRegs AS RegType
CALL INTERRUPT(&H11, InRegs, OutRegs)
GetEquipment% = OutRegs.ax
```

Dopo aver restituito questa parola di 16 bit al programma chiamante, si possono decodificare i suoi bit secondo la tabella precedente (oppure si può adattare la funzione in modo che li interpreti automaticamente).
Il listato 9.2 mostra la funzione completa.

```
DECLARE FUNCTION GetEquipment% ()

TYPE RegType
        ax      AS INTEGER
        bx      AS INTEGER
        cx      AS INTEGER
        dx      AS INTEGER
        bp      AS INTEGER
        si      AS INTEGER
        di      AS INTEGER
        flags   AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs AS RegType)

FUNCTION GetEquipment%

REM Restituisce questa parola, bit per bit:
REM Bit      Significato

REM ------------------------------------------------------------

REM 15,14    Numero di stampanti installate
REM 13       È installato un modem interno? (1 = sì)
REM 12       Non utilizzato
REM 11,10,9  Numero di schede RS-232 collegate
REM 8        Non utilizzato
REM 7,6      Numero di drive per floppy disk: 00 = 1 drive,
             01 = 2 drive
REM 5,4      Tipo di video: 01 = 40x25 a colori, 10 = 80x25
             a colori, 11 = 80x25 monocromatico
REM 3        Non utilizzato
REM 2        È installato un mouse? (1 = sì)
REM 1        È installato un coprocessore matematico? (1 = sì)
REM 0        Sono installati drive per floppy disk? (1 = sì)

   DIM InRegs AS RegType, OutRegs AS RegType
   CALL INTERRUPT(&H11, InRegs, OutRegs)
   GetEquipment% = OutRegs.ax

END FUNCTION
```

Listato 9.2. *GetEquipment%: restituisce informazioni sull'hardware installato.*

Questo programma, per esempio, controlla se sia o meno installato un coprocessore matematico:

```
REM Esempio di uso di GetEquipment%

DECLARE FUNCTION GetEquipment% ( )

Check% = GetEquipment%

IF (Check% AND 2) THEN
     PRINT "Hai un coprocessore matematico."
ELSE
     PRINT "Non hai un coprocessore matematico."
END IF
```

Come si può vedere, gli interrupt del BIOS e del DOS hanno davvero qualcosa di utile da offrire a chi programma in BASIC e, attraverso INTERRUPT( ) e INTERRUPTX( ), possono essere sfruttati adeguatamente.

> ***Consiglio:*** Andando a controllare il bit 2 del valore restituito da questo servizio si scopre se sia installato o meno un mouse (1 = sì). Si tratta di un buon test da inserire prima di utilizzare il programma per il mouse sviluppato nel capitolo 3.

Continuando nella panoramica del BIOS e del DOS, l'interrupt &H12 originariamente era stato progettato per determinare le dimensioni della memoria installata, ma restituisce solamente la quantità di memoria installata sulla scheda madre, senza considerare le eventuali schede aggiuntive, il che significa che non è più un servizio affidabile.
L'interrupt successivo, &H13, è ricco di servizi: gestisce al livello più basso i drive per dischetti e dischi rigidi. Poiché i dati su disco sono sempre molto preziosi, in questo libro l'interrupt &H13 non verrà utilizzato, ma in effetti mette a disposizione molti strumenti davvero potenti:

Interrupt &H13	Servizio 0	Reset del disco
Interrupt &H13	Servizio 1	Legge lo stato dell'ultima operazione
Interrupt &H13	Servizio 2	Legge settori in memoria
Interrupt &H13	Servizio 3	Scrive settori su disco
Interrupt &H13	Servizio 4	Verifica settori
Interrupt &H13	Servizio 8	Restituisce i parametri del drive
Interrupt &H13	Servizio &HA	Riservato
Interrupt &H13	Servizio &HB	Riservato
Interrupt &H13	Servizio &HC	Cerca
Interrupt &H13	Servizio &HD	Reset alternativo del disco
Interrupt &H13	Servizio &HE	Riservato
Interrupt &H13	Servizio &HF	Riservato
Interrupt &H13	Servizio &H10	Controlla se il drive è pronto
Interrupt &H13	Servizio &H11	Ricalibra il drive del disco rigido

Interrupt &H13	Servizi &H12-&H14	Servizi diagnostici
Interrupt &H13	Servizio &H19	Parcheggia le testine (solo PS/2)

I due interrupt successivi hanno a che fare soprattutto con l'I/O. Per esempio, l'interrupt &H14 è quello che gestisce la porta seriale:

Interrupt &H14	AH = 0	Inizializza la porta RS232
Interrupt &H14	AH = 1	Invia un carattere attraverso la porta seriale
Interrupt &H14	AH = 2	Riceve un carattere dalla porta seriale
Interrupt &H14	AH = 3	Restituisce lo stato della porta seriale

L'interrupt &H15 gestisce la porta per il registratore a cassette (ormai del tutto obsoleta) che era installata sui primi PC:

Interrupt &H15	I/O registratore a cassette

L'interrupt &H16 gestisce i dati della tastiera: consente di leggere i tasti premuti (mentre l'interrupt 9 è quello di basso livello che legge i bit direttamente dalla tastiera e li decodifica per trovare i caratteri):

Interrupt &H16	Servizio 0	Legge un tasto dalla tastiera
Interrupt &H16	Servizio 1	Controlla se il tasto è pronto per essere letto
Interrupt &H16	Servizio 2	Determina lo stato della tastiera

L'interrupt &H17 gestisce la stampante:

Interrupt &H17	Servizio 0	Stampa il carattere in al
Interrupt &H17	Servizio 1	Inizializza la porta della stampante
Interrupt &H17	Servizio 2	Legge in ah lo stato della stampante

L'interrupt &H18 è l'interrupt residente del ROM BASIC. I primi PC avevano in ROM una versione ridotta del BASIC, cosicché, anche accendendo la macchina senza dischetto di sistema, la macchina era comunque in grado di fare qualcosa: veniva lanciata la versione interna del BASIC. Se si esegue l'interrupt &H18 in quel tipo di macchine, si avvia il ROM BASIC.

Interrupt &H18	BASIC residente

L'interrupt &H19 è quello di boot, cioè l'interrupt che viene eseguito quando si accende il computer. Chiamandolo, si può riavviare la macchina (un metodo un po' drastico per concludere un programma):

Interrupt &H19	Bootstrap

L'interrupt &H1A gestisce l'ora (è l'interrupt utilizzato dai comandi TIME e DATE del DOS):

Interrupt &H1A	Servizio 0	Legge l'ora del giorno
Interrupt &H1A	Servizio 1	Imposta l'ora del giorno

Gli altri interrupt del BIOS, fatta eccezione per &H1C, non sono routine software, bensì contengono varie tabelle e indirizzi. Per esempio, l'interrupt &H1B contiene l'indirizzo a cui viene trasferito il controllo se si preme Ctrl-Pausa e l'interrupt &HID contiene i parametri correnti del video. L'interrupt &H1E contiene l'indirizzo dei parametri dei dischetti e, se si sa quello che si fa, si possono cambiare quei valori (fra l'altro i tempi di ricerca e di inizializzazione per accelerare queste operazioni). L'interrupt &H1F può contenere l'indirizzo dei caratteri grafici utilizzati sullo schermo.

Interrupt &H1B	Indirizzo della pausa da tastiera
interrupt &H1C	Interrupt del "tic" di orologio
Interrupt &H1D	Tabelle dei parametri del video
Interrupt &H1E	Parametri dei dischetti
Interrupt &H1F	Definizioni dei caratteri grafici

L'interrupt &H1C è quello del timer e viene chiamato tutto le volte che il timer interno incrementa il suo conteggio (18,2 volte al secondo). Molti programmi residenti sfruttano questo interrupt per assumere temporaneamente il controllo della macchina.

***Nota**: I programmi di tipo TSR assumono il controllo modificando l'indirizzo interno di questo interrupt, in modo che punti al loro codice in memoria. In questo modo, a ogni "tic" del timer viene eseguito il codice del TSR.*

Con questo si conclude la rassegna degli interrupt del BIOS: è ora di passare al DOS.

GLI INTERRUPT DEL DOS

Gli interrupt del DOS vanno da &H20 a &H3E. Il primo, &H20, è chiamato a livello di linguaggio assembly quando un programma ha concluso la sua attività e vuol restituire il controllo al DOS. È l'interrupt che riporta il prompt C:\> sullo schermo:

Interrupt &H20	Conclusione di programma

Con l'introduzione di altri metodi di conclusione dei programmi, questo interrupt è diventato un po' obsoleto. Un altro metodo di conclusione dei programmi è l'interrupt &H21, servizio &H4C, che consente la restituzione di un codice d'errore per indicare il successo o il fallimento.
Si arriva così all'interrupt più complesso, &H21, nei cui servizi sono racchiuse quasi tutte le capacità del DOS: gestione dei file, lettura della tastiera, visualizzazione sullo schermo. È l'interrupt utilizzato da tutti i comandi DOS (come COPY, FORMAT o DIR).

Qualcuno avrà notato che si può visualizzare sullo schermo e leggere dalla tastiera anche con alcuni servizi del BIOS, ed è vero; ma i servizi del DOS sono molto più sofisticati rispetto ai servizi di basso livello del BIOS e, in effetti, molti servizi del DOS richiamano gli stessi servizi del BIOS per il controllo di basso livello.
Sfortunatamente, i servizi dell'interrupt &H21 non sembrano organizzati in un ordine ben definito. In linea di massima, comunque, i primi hanno a che fare con l'immissione e l'emissione di caratteri (fatta eccezione per il servizio 0, che fa la stessa cosa dell'interrupt &H20):

Interrupt &H21	Servizio 0	Conclusione di programma
Interrupt &H21	Servizio 1	Immissione da tastiera
Interrupt &H21	Servizio 2	Emissione di carattere sullo schermo
Interrupt &H21	Servizio 3	Immissione da dispositivo ausiliario standard
Interrupt &H21	Servizio 4	Emissione a dispositivo ausiliario standard
Interrupt &H21	Servizio 5	Uscita a stampante
Interrupt &H21	Servizio 6	I/O su console
Interrupt &H21	Servizio 7	Immissione su console senza eco
Interrupt &H21	Servizio 8	Immissione su console senza eco ma con verifica di ^C
Interrupt &H21	Servizio 9	Stampa di stringa di caratteri
Interrupt &H21	Servizio &HA	Immissione di stringa
Interrupt &H21	Servizio &HB	Verifica dello stato di immissione
Interrupt &H21	Servizio &HC	Svuota il buffer di tastiera

Abbiamo già visto all'opera uno o due di questi servizi. I prossimi due invece permettono di reinizializzare un disco o di modificare il disco di default:

Interrupt &H21	Servizio &HD	Reinizializzazione del disco
Interrupt &H21	Servizio &HE	Selezione di disco

Nel BASIC PDS, per impostare il drive di default si può utilizzare l'istruzione CHDRIVE. Quest'ultima, però, non è disponibile in QuickBASIC, perciò val la pena di scrivere una nuova funzione, SetDefaultDisk%( ), per gli utenti di QuickBASIC. Il servizio &H3 dell'interrupt &H21 consente di modificare il drive di default secondo necessità.

IMPOSTAZIONE DEL DISCO DI DEFAULT

La funzione che ora si scriverà, SetDefaultDisk%( ), consentirà di impostare il disco di default. È bene fare in modo che questa funzione restituisca 0 se vi è qualche problema (se, per esempio, non esiste il disco destinazione) e 1 in caso di successo. Ecco come si potrebbe usare SetDefaultDisk%( ):

```
Drive$ = "A"
Check% = SetDefaultDisk% (Drive%)
IF Check% = 0 THEN PRINT "Errore."
```

Qui si può usare il servizio &HE dell'interrupt &H21. Innanzitutto si può controllare il parametro Drive$, per assicurarsi che sia lungo un solo carattere:

```
SetDefaultDisk% = 0

IF LEN(Drive$) <> 1 THEN EXIT FUNCTION
   :
```

Il servizio &HE attende il nuovo numero di drive (0 per A, 1 per B ecc.) nel registro dl, perciò si può procedere in questo modo:

```
SetDefaultDisk% = 0

IF LEN(Drive$) <> 1 THEN EXIT FUNCTION

InRegs.ax = &HE00
InRegs.dx = ASC(UCASE$(Drive$)) - ASC("A")
CALL INTERRUPT(&H21, InRegs, OutRegs)
   :
```

Poi si può controllare il nuovo numero di drive (restituito in al) con il numero di drive richiesto (che si trova ancora in dl) per essere sicuri che la modifica sia stata effettivamente apportata.

```
SetDefaultDisk% = 0

IF LEN(Drive$) <> 1 THEN EXIT FUNCTION

InRegs.ax = &HE00
InRegs.dx = ASC(UCASE$(Drive$)) - ASC("A")
CALL INTERRUPT(&H21, InRegs, OutRegs)
IF (InRegs.dx AND &HFF) <> (InRegs.ax AND &HFF) THEN EXIT
FUNCTION

SetDefaultDisk% = 1
```

Se il drive corrente non è uguale a quello richiesto, si esce, lasciando SetDefaultDisk a 0 per indicare fallimento. Se i due drive coincidono, si imposta SetDefaultDisk% a 1 e si esce.
Il listato 9.3 mostra la funzione completa.

```
DECLARE FUNCTION SetDefaultDisk% (Drive$)

FUNCTION SetDefaultDisk% (Drive$)

   DIM InRegs AS RegType, OutRegs AS RegType

   SetDefaultDisk% = 0

   IF LEN(Drive$) <> 1 THEN EXIT FUNCTION

   InRegs.ax = &HE00
   InRegs.dx = ASC(UCASE$(Drive$)) - ASC("A")
   CALL INTERRUPT(&H21, InRegs, OutRegs)
   IF (InRegs.dx AND &HFF) <> (InRegs.ax AND &HFF)
      THEN EXIT FUNCTION

   SetDefaultDisk% = 1

END FUNCTION
```

Listato 9.3. *La funzione SetDefaultDisk% (Drive$).*

Continuiamo con la nostra rassegna del DOS. Il gruppo di servizi che segue riguarda prevalentemente i file. Si tratta però di servizi in gran parte obsolete, che è bene evitare di utilizzare:

Interrupt &H21	Servizio &HF	Apre un file preesistente
Interrupt &H21	Servizio &H10	Chiude il file
Interrupt &H21	Servizio &H11	Cerca il primo file che soddisfi i requisiti
Interrupt &H21	Servizio &H12	Cerca il successivo file che soddisfi i requisiti
Interrupt &H21	Servizio &H13	Cancella file
Interrupt &H21	Servizio &H14	Lettura sequenziale
Interrupt &H21	Servizio &H15	Scrittura sequenziale
Interrupt &H21	Servizio &H16	Creazione di file
Interrupt &H21	Servizio &H17	Ridenominazione di file

***Nota**: Questi servizi lavorano con i blocchi di controllo file, che non possono gestire i nomi di percorso (erano utilizzati soprattutto prima della disponibilità di dischi rigidi). I servizi aggiornati per i file iniziano con &H3C.*

Altri servizi per i dischi:

Interrupt &H21	Servizio &H18	Interno al DOS
Interrupt &H21	Servizio &H19	Trova il disco di default
Interrupt &H21	Servizio &H1A	Imposta la locazione DTA

Interrupt &H21	Servizio &H1B	Informazioni della FAT per il drive di default
Interrupt &H21	Servizio &H1C	Informazioni della FAT per il drive specificato
Interrupt &H21	Servizi &H1D-1H20	Interni al DOS

Alcuni di questi servizi sono molto utili. In effetti, si può subito sfruttarne uno, il servizio &H19, che restituisce il disco di default.

COME OTTENERE IL DISCO DI DEFAULT

Si può scrivere una funzione GetDefaultDisk$( ) che restituisca una stringa di un carattere per indicare la lettera del drive corrente:

```
DriveNow% = GetDefaultDisk$ ( )
```

Questa funzione farà uso del servizio &H19 del servizio &H21 per determinare quale sia il disco di default.

> ***Consiglio:*** Il servizio &H19 dell'interrupt &H21 è utile se si è appena trasferito il controllo a un sottoprogramma e si vuole scoprire se si sta lavorando con un disco rigido o un dischetto.

Per richiedere il servizio &H19, si carica &H19 in ah e si chiama l'interrupt &H21. Il numero del disco di default viene restituito in al. Qui, 0 corrisponde al drive A, 1 al B e via elencando. Il numero in al viene convertito in una lettera di drive in questo modo:

```
DIM InRegs AS RegType, OutRegs AS RegType
InRegs.ax = &H1900
CALL INTERRUPT(&H21, InRegs, OutRegs)
GetDefaultDisk$ = CHR$(OutRegs.ax AND &HFF) + ASC("A"))
```

E questo è tutto. Il listato 9.4 mostra la funzione completa.

```
DECLARE FUNCTION GetDefaultDisk$ ( )
TYPE RegType
        ax        AS INTEGER
        bx        AS INTEGER
        cx        AS INTEGER
        dx        AS INTEGER
        bp        AS INTEGER
        si        AS INTEGER
```

continua

Listato 9.4. *La funzione GetDefaultDisk$*

```
        di        AS INTEGER
        flags     AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs as RegType)

FUNCTION GetDefaultDisk$

   DIM InRegs AS RegType, OutRegs AS RegType
   InRegs.ax = &H1900
   CALL INTERRUPT(&H21, InRegs, OutRegs)
   GetDefaultDisk$ = CHR$(OutRegs.ax AND &HFF) + ASC("A"))

END FUNCTION
```

Listato 9.4. *La funzione GetDefaultDisk$.*

Ed ecco un programmino che mette all'opera GetDefaultDisk$:

```
REM Esempio di uso di GetDefaultDisk$

DECLARE FUNCTION GetDefaultDisk$ ( )

PRINT "Il disco di default è: "; GetDefaultDisk$
```

È un programma brevissimo e ben definito, che si limita a stampare la lettera identificativa del drive di default, restituita da GetDefaultDisk$.
I prossimi servizi del DOS riguardano principalmente l'I/O su file, e fanno parte di quel gruppo di servizi DOS ormai obsoleti che è bene evitare. Sono i servizi di I/O per i file basati sui blocchi di controllo.

Interrupt &H21	Servizio &H21	Lettura casuale
Interrupt &H21	Servizio &H22	Scrittura casuale
Interrupt &H21	Servizio &H23	Dimensioni del file
Interrupt &H21	Servizio &H24	Imposta campo record casuale
Interrupt &H21	Servizio &H25	Imposta il vettore degli interrupt
Interrupt &H21	Servizio &H26	Crea un nuovo segmento di programma (PSP)
Interrupt &H21	Servizio &H27	Lettura casuale di blocco
Interrupt &H21	Servizio &H28	Scrittura casuale di blocco
Interrupt &H21	Servizio &H29	Analizza un nome di file

Poi segue una raccolta di servizi che non hanno proprio niente in comune:

Interrupt &H21	Servizio &H2A	Ottiene la data
Interrupt &H21	Servizio &H2B	Imposta la data

Interrupt &H21	Servizio &H2C	Ottiene l'ora
Interrupt &H21	Servizio &H2D	Imposta l'ora
Interrupt &H21	Servizio &H2E	Imposta o reinizializza il commutatore di verifica
Interrupt &H21	Servizio &H2F	Restituisce l'indirizzo dell'area di trasferimento corrente
Interrupt &H21	Servizio &H30	Ottiene il numero di versione del DOS
Interrupt &H21	Servizio &H31	Termina il processo e lo mantiene residente
Interrupt &H21	Servizio &H32	Interno al DOS
Interrupt &H21	Servizio &H33	Verifica di Control-Pausa
Interrupt &H21	Servizio &H34	Interno al DOS
Interrupt &H21	Servizio &H35	Ottiene il vettore degli interrupt

Il gruppo successivo di servizi ha di nuovo a che fare soprattutto con i dischi:

Interrupt &H21	Servizio &H36	Determina lo spazio libero su disco
Interrupt &H21	Servizio &H37	Interno al DOS
Interrupt &H21	Servizio &H38	Restituisce informazioni dipendenti dalla nazionalizzazione
Interrupt &H21	Servizio &H39	Crea una sottodirectory
Interrupt &H21	Servizio &H3A	Cancella una sottodirectory
Interrupt &H21	Servizio &H3B	Cambia la directory corrente

Uno di questi (il servizio &H36 dell'interrupt &H21, che determina lo spazio libero su disco) è particolarmente utile per chi programma in BASIC. Si può sfruttarlo per sviluppare una funzione.

DETERMINARE LO SPAZIO LIBERO

Qui si costruirà una funzione in grado di restituire le dimensioni (in byte) dello spazio disponibile su un disco indicato dall'utente. La funzione può essere chiamata DiskFree&( ) e verrà usata in questo modo:

```
BytesLeft& = DiskFree& (Drive$)
```

> ***Consiglio***: DiskFree&( ) può risultare estremamente utile se si deve creare un file di grandi dimensioni e si vuole controllare innanzitutto lo spazio disponibile sul disco destinazione.

Qui Drive$ è la lettera che corrisponde al drive che si vuol controllare (C, per esempio). Poiché è possibile che il drive specificato non esista, bisogna fare in modo che DiskFree& restituisca un valore -1 se si verifica un errore.
Per cominciare, bisogna caricare &H36 in ah:

```
DIM InRegs AS RegType, OutRegs AS RegType

DiskFree& = -1
IF LEN(Drive$) <> 1 THEN EXIT FUNCTION

InRegs.ax = &H3600
    :
    :
```

Bisogna poi scrivere il numero del drive in dl (gli otto bit meno significativi del registro dx). Per questo servizio (e a differenza di quanto accade con il servizio &H19), il numero del drive è 1 per il drive A, 2 per il B e così via. In altre parole, ASC(UCASE$(Drive$)) - ASC("A") + 1. Poi si chiama INTERRUPT( ):

```
DIM InRegs AS RegType, OutRegs AS RegType

DiskFree& = -1
IF LEN(Drive$) <> 1 THEN EXIT FUNCTION

InRegs.ax = &H3600
InRegs.dx = ASC(UCASE$(Drive$)) - ASC("A") + 1
CALL INTERRUPT(&H21, InRegs, OutRegs)
    :
    :
```

Se questo servizio colloca un -1 in ax, allora il drive di destinazione non esiste o non risponde. In questo caso, al ritorno DiskFree& deve avere il valore -1:

```
DIM InRegs AS RegType, OutRegs AS RegType

DiskFree& = -1
IF LEN(Drive$) <> 1 THEN EXIT FUNCTION

InRegs.ax = &H3600
InRegs.dx = ASC(UCASE$(Drive$)) - ASC("A") + 1
CALL INTERRUPT(&H21, InRegs, OutRegs)

IF OutRegs.ax = -1 THEN EXIT FUNCTION
    :
    :
```

Altrimenti, si deve effettuare il prodotto ax * bx * cx per ottenere il numero dei byte liberi sul disco. Per evitare overflow, innanzitutto si carica ax in una variabile di tipo LONG, poi si esegue il prodotto:

```
Dim InRegs AS RegType, OutRegs AS RegType
DiskFree& = -1
IF LEN(Drive$) <> 1 THEN EXIT FUNCTION

InRegs.ax = &H3600
InRegs.dx = ASC(UCASE$(Drive$)) - ASC("A") + 1
CALL INTERRUPT(&H21, InRegs, OutRegs)

IF OutRegs.ax = -1 THEN EXIT FUNCTION

Temp& = OutRegs.ax
DiskFree& = Temp& * OutRegs.bx * OutRegs.cx
```

***Nota**: bx contiene il numero delle unità di allocazione (cluster) disponibili, ax il numero dei settori per unità di allocazione e cx il numero dei byte per settore. Il prodotto dei tre pertanto è il numero dei byte liberi disponibili sul disco.*

Ed è fatta: ora il valore risultante viene restituito come valore di DiskFree&. Il listato 9.5 mostra la funzione completa.

```
DECLARE FUNCTION DiskFree& (Drive$)
TYPE RegType
        ax        AS INTEGER
        bx        AS INTEGER
        cx        AS INTEGER
        dx        AS INTEGER
        bp        AS INTEGER
        si        AS INTEGER
        di        AS INTEGER
        flags     AS INTEGER
END TYPE

DECLARE SUB INTERRUPT (IntNo AS INTEGER, InRegs AS RegType,
                       OutRegs as RegType)


FUNCTION DiskFree& (Drive$)

   Dim InRegs AS RegType, OutRegs AS RegType
  DiskFree& = -1
   IF LEN(Drive$) <> 1 THEN EXIT FUNCTION

   InRegs.ax = &H3600
```

continua

Listato 9.5 *DiskFree& - Riporta spazio disponibile su disco*

```
    InRegs.dx = ASC(UCASE$(Drive$)) - ASC("A") + 1
   CALL INTERRUPT(&H21, InRegs, OutRegs)
   IF OutRegs.ax = -1 THEN EXIT FUNCTION

   Temp& = OutRegs.ax
   DiskFree& = Temp& * OutRegs.bx * OutRegs.cx

END FUNCTION
```

Listato 9.5 *DiskFree& - Riporta spazio disponibile su disco*

Ed ecco un programmino che mette all'opera DiskFree&:

```
REM Esempio di uso di DiskFree&

DECLARE FUNCTION DiskFree& (Drive$)

MyDrive$ = "C"

PRINT "Lo spazio libero sul disco "; MyDrive$; "è pari a ";
      DiskFree&(MyDrive$)
```

Questo programmino semplicemente va a controllare il disco C e riferisce quanti sono i byte liberi. Non deve esser eseguito se non si ha a disposizione un disco C; senza un drive da controllare effettivamente, i risultati sono senza senso.
Continuando nella rassegna del DOS, si arriva ai servizi aggiornati per la gestione dei file.

I SERVIZI DEL DOS PER I FILE

A partire dalla versione 2.0, il DOS ha consentito l'utilizzo di percorsi (PATH) per la ricerca dei file e questo ha reso obsoleti i precedenti servizi. I nuovi servizi, che utilizzano gli *handle di file*, iniziano qui. Gli handle di file funzionano sostanzialmente come gli handle di finestra che sono stati sviluppati nel capitolo 2. Un handle di file è semplicemente una word che identifica uno specifico file aperto. Ecco i nuovi servizi per i file:

Interrupt &H21	Servizio &H3C	Creazione di file
Interrupt &H21	Servizio &H3D	Apertura di file
Interrupt &H21	Servizio &H3E	Chiusura di handle di file
Interrupt &H21	Servizio &H3F	Lettura da file o device
Interrupt &H21	Servizio &H40	Scrittura su file o device
Interrupt &H21	Servizio &H41	Cancellazione di file
Interrupt &H21	Servizio &H42	Sposta il puntatore lettura/scrittura
Interrupt &H21	Servizio &H43	Modifica di attributo di file

Interrupt &H21	Servizio &H44	Controllo I/O
Interrupt &H21	Servizio &H45	Duplicazione di handle di file
Interrupt &H21	Servizio &H46	Duplicazione forzata di handle di file
Interrupt &H21	Servizio &H47	Ottiene la directory corrente su un disco specificato

I successivi servizi del DOS riguardano l'allocazione o la deallocazione di memoria, e sono importanti soprattutti in ambienti multiutente (di solito il DOS assegna a un programma tutta la memoria disponibile, a meno che la memoria sia condivisa):

Interrupt &H21	Servizio &H48	Alloca memoria
Interrupt &H21	Servizio &H49	Libera memoria allocata
Interrupt &H21	Servizio &H4A	SETBLOCK (allocazione di memoria)
Interrupt &H21	Servizio &H4B	Carica o esegue un programma - EXEC
Interrupt &H21	Servizio &H4C	Uscita con codice di ritorno
Interrupt &H21	Servizio &H4D	Ottiene il codice di ritorno di un sottoprocesso

I due servizi seguenti, però, possono essere molto utili anche per chi programma in BASIC:

Interrupt &H21	Servizio &H4E	Trova il primo file che soddisfa i requisiti
Interrupt &H21	Servizio &H4F	Trova il successivo file che soddisfa i requisiti

Il servizio &H4E cerca in una directory il primo nome di file che soddisfi una specifica di file indicata dall'utente. Il servizio &H4F continua l'opera iniziata da &H4E: usandolo varie volte trova tutti gli altri file che soddisfano i requisiti (il servizio &H4E esegue dei compiti di inizializzazione che debbono essere svolti una sola volta). Si possono aggiungere al BASIC anche queste capacità.

TROVARE IL PRIMO FILE CHE CORRISPONDE AI REQUISITI

Il servizio &H4E è un servizio importante e ci introdurrà alle complessità dell'interfacciamento con i servizi del DOS per i file. Questo servizio consente di cercare all'interno di una directory un file che corrisponda a una specifica data e lo si può usare in una funzione a cui si potrà dare il nome GetFirstMatchingFile$( ). Per esempio, per trovare un file corrispondente alla specifica PROGRAM.* nella sottodirectory C:\ARCHIVI, la si usa in questo modo:

```
Match$ = GetFirstMatchingFile$ ("C:\ARCHIVI\PROGRAM.*")
```

Oppure, per vedere se nella directory vi sono o meno dei file:

```
Match$ = GetFirstMatchingFile$ ("C:\ARCHIVI\*.*")
```

Si può progettare la funzione in modo che restituisca una stringa costituita dal nome del primo file che corrisponde alla specifica (il nome di percorso verrà omesso perché il servizio &H4E non lo restituisce).

> ***Consiglio:*** Con questa funzione si possono cercare file in qualunque parte del disco, si può controllare se esiste un file prima di sovrascriverlo, oppure si possono esaminare i contenuti di intere sottodirectory.

Poiché il servizio &H4E restituisce solo il nome del primo file che corrisponde alla specifica, è necessario utilizzare un'altra funzione, che si potrà chiamare GetNextMatchingFile$( ), per continuare la ricerca. Per cercare più file in una directory, si usa prima GetFirstMatchingFile$( ); se restituisce un nome di file (e non una stringa nulla, "", che indica l'assenza di file corrispondenti alla specifica), si usa poi GetNextMatchingFile$( ), chiamandola tante volte finché non si esauriscono le corrispondenze (finché cioè non viene restituito "").
Passiamo a scrivere la funzione GetFirstMatchingFile$( ). Il servizio &H4E dell'interrupt &H21 restituisce le informazioni relative ai file corrispondenti nella *Disk Transfer Area* (area trasferimento disco, o DTA). DTA è un buffer riservato per le manipolazioni di file a basso livello da parte del DOS (ed è quasi completamente obsoleto, fatta eccezione per questo servizio). Per questo, è necessario trovare l'indirizzo della DTA corrente, cosa che consente proprio uno dei servizi visti poco più sopra (il servizio &H2F dell'interrupt &H21). Come si è visto nel capitolo 5, nel PC gli indirizzi sono costituiti da due word, l'indirizzo di un segmento e l'indirizzo di un offset. Il servizio &H2F restituisce ambedue gli indirizzi della DTA: li inseriremo in DTASegment% e DTAOffset%, rispettivamente:

```
FUNCTION GetFirstMatchingFile$ (FileSpec$)

   DIM InRegs AS RegTypeX, OutRegs AS RegTypeX
   REM Ottiene l'indirizzo della Disk Transfer Area (DTA

   InRegs.ax = &H2F00
   CALL INTERRUPTX(&H21, InRegs, OutRegs)
   DTASegment% = OutRegs.es
   DTAOffset% = OutRegs.bx
        :
        :
```

Ora che si dispone dell'indirizzo della DTA, si può cercare il primo file che corrisponde alle specifiche. Consultando l'appendice su BIOS e DOS, si può vedere che si deve passare all'interrupt &H21 una stringa ASCII che contenga la specifica del

file. Una stringa ASCIIZ è semplicemente una stringa di caratteri con un carattere terminatore ASCII 0, cioè CHR$(0), alla fine.
Se la specifica di file che ci viene passata è FileSpec$ (per esempio, questa stringa conterrà *.DAT oppure *.*) se ne può costruire una stringa ASCIIZ in questo modo: NameFile$ = FileSpec$ + CHR$(0). Ora si deve passare l'indirizzo di NameFile$ all'interrupt &H21. Si può usare VARSEG( ) per ottenere l'indirizzo di segmento di una stringa, e si userà la funzione BASIC SADD( ) (e non VARPTR( )) per ottenerne l'indirizzo di offset.
L'indirizzo di segmento va in un nuovo registro, ds (che sta per "data segment", segmento dei dati) e l'indirizzo dell'offset va in dx. Il registro ds è uno dei quattro *registri di segmento* nel PC (figura 9.1). Questi registri sono progettati per tener traccia degli indirizzi di segmento. Esistono quattro registri di segmento: cs, ds, es e ss e tutti sono utilizzati per contenere indirizzi di segmento. Il registro ds contiene l'indirizzo del segmento dati corrente; es contiene l'indirizzo del segmento extra; ss quello del segmento dello stack e cs quello del segmento del codice. Verranno utilizzati più approfonditamente nel capitolo 11.

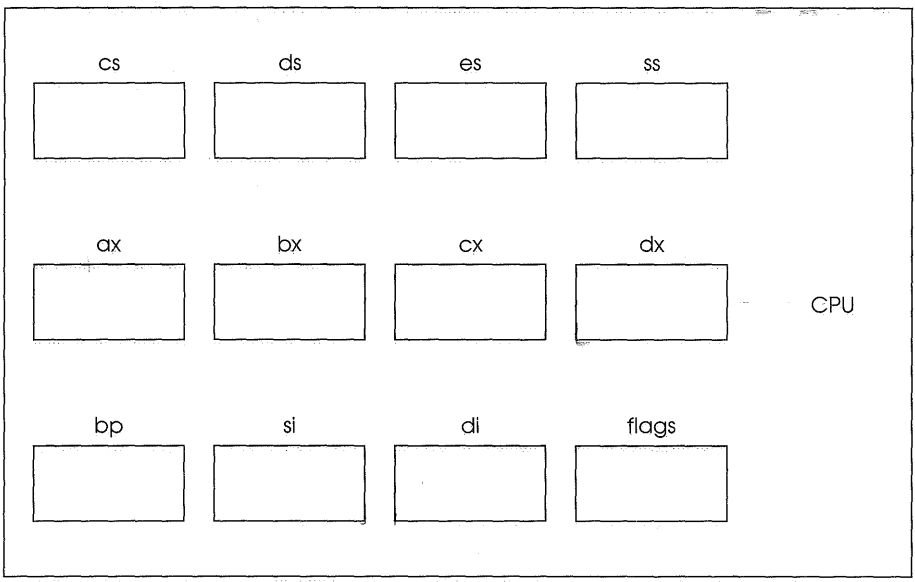

Figura 9.1

Si può caricare in ds e dx l'indirizzo di NameFile$, in questo modo:

```
FUNCTION GetFirstMatchingFile$ (FileSpec$)

    DIM InRegs AS RegTypeX, OutRegs AS RegTypeX
    REM Ottiene l'indirizzo della Disk Transfer Area (DTA
```

```
InRegs.ax = &H2F00
CALL INTERRUPTX(&H21, InRegs, OutRegs)
DTASegment% = OutRegs.es
DTAOffset% = OutRegs.bx

REM Ora si trova la prima corrispondenza

NameFile$ = FileSpec$ + CHR$(0)
InRegs.ds = VARSEG(NameFile$)
InRegs.dx = SADD(NameFile$)
     :
```

Ora si deve decidere che tipo di file si vuol trovare; cioè, qual è l'*attributo di file* da utilizzare nella ricerca. Il DOS usa valori di attributo di file come questi (e si può consultare questa lista se si vuole usare qualsiasi altro servizio del DOS per i file):

Attributo	Significato
0	File normale
1	File a sola lettura
2	File nascosto
4	File di sistema
8	Etichetta di volume di un disco
16	Nome di sottodirectory

Consiglio: Utilizzando questa funzione e impostando a 2 l'attributo di file, si possono cercare sul disco file nascosti. Si possono anche cercare i file di sistema IBMDOS.COM o MSDOS.COM, per stabilire se il disco corrente è un disco di sistema.

In altre parole, se si passa un attributo di file pari a 1, saranno presi in considerazione esclusivamente i file a sola lettura. Se si usa un attributo 0, si prendono in considerazione solo i file normali. L'attributo di file deve essere passato in cx, perciò ai fini dell'esempio si collochi 0 in cx e si chiami l'interrupt &H21:

```
FUNCTION GetFirstMatchingFile$ (FileSpec$)

   DIM InRegs AS RegTypeX, OutRegs AS RegTypeX
   REM Ottiene l'indirizzo della Disk Transfer Area (DTA)

   InRegs.ax = &H2F00
   CALL INTERRUPTX(&H21, InRegs, OutRegs)
   DTASegment% = OutRegs.es
   DTAOffset% = OutRegs.bx

   REM Ora si trova la prima corrispondenza
```

```
NameFile$ = FileSpec$ + CHR$(0)
InRegs.ds = VARSEG(NameFile$)
InRegs.dx = SADD(NameFile$)
InRegs.cx = 0
InRegs.ax = &H4E00
CALL INTERRUPTX(&H21, InRegs, OutRegs)
      :
      :
```

Quando si ritorna dal servizio &H4E dell'interrupt &H21, gli esiti possibili sono due: o si trova un file che corrisponde alle specifiche, oppure non lo si trova. Se non ci sono corrispondenze questo servizio imposta a 1 il flag carry.
Il flag carry è uno di otto flag interni all'80x86, che abbiamo già incontrato nel capitolo 1:

Bit di OutRegs.flag	Flag
11	Flag di overflow
10	Flag di direzione
9	Flag di abilitazione interrupt
8	Flag di intercettazione
7	Flag di segno
6	Flag zero
4	Flag di carry ausiliario
2	Flag di parità
0	Flag di carry

Come si può vedere, il flag di carry è nel bit 0 di OutRegs.flags; se quel bit è 1, il flag di carry è stato impostato. Si può leggere il valore del flag di carry dopo la chiamata a INTERRUPTX( ) esaminando il valore di OutRegs.flags AND 1. Se non è stata trovata alcuna corrispondenza, il flag di carry sarà impostato (OutRegs.flags AND 1 sarà uguale a 1), perciò si deve restituire una stringa nulla, "", in questo modo:

```
FUNCTION GetFirstMatchingFile$ (FileSpec$)

   DIM InRegs AS RegTypeX, OutRegs AS RegTypeX
   REM Ottiene l'indirizzo della Disk Transfer Area (DTA

   InRegs.ax = &H2F00
   CALL INTERRUPTX(&H21, InRegs, OutRegs)
   DTASegment% = OutRegs.es
   DTAOffset% = OutRegs.bx

   REM Ora si trova la prima corrispondenza
   NameFile$ = FileSpec$ + CHR$(0)
   InRegs.ds = VARSEG(NameFile$)
   InRegs.dx = SADD(NameFile$)
   InRegs.cx = 0
   InRegs.ax = &H4E00
```

```
CALL INTERRUPTX(&H21, InRegs, OutRegs)

IF OutRegs.flags AND 1 THEN
 GetFirstMatchingFile$ = ""
 EXIT FUNCTION
END IF
     :
```

Invece, se *si trova* una corrispondenza, la DTA sarà riempita come indicato in figura 9.2 (si veda anche l'Appendice).

Inizio DTA:	
	21 byte: riservati.
	1 byte: attribuito del file trovato
	2 byte: ora del file trovato
	2 byte: data del file trovato
	2 byte: parola bassa delle dimensioni del file
	2 byte: parola alta delle dimensioni del file
	13 byte: nome del file trovato, in forma ASCIIZ
Fine DTA:	

Figura 9.2

Supponiamo che ci sia un file che corrisponde alle specifiche. In questo caso il nome del file occuperà 29 byte nella DTA, sotto forma di stringa ASCIIZ. Si può collocare l'indirizzo dell'offset in una variabile MatchOffset%, in questo modo:

```
FUNCTION GetFirstMatchingFile$ (FileSpec$)

   DIM InRegs AS RegTypeX, OutRegs AS RegTypeX
   REM Ottiene l'indirizzo della Disk Transfer Area (DTA

   InRegs.ax = &H2F00
   CALL INTERRUPTX(&H21, InRegs, OutRegs)
   DTASegment% = OutRegs.es
   DTAOffset% = OutRegs.bx

   REM Ora si trova la prima corrispondenza

   NameFile$ = FileSpec$ + CHR$(0)
   InRegs.ds = VARSEG(NameFile$)
   InRegs.dx = SADD(NameFile$)
   InRegs.cx = 0
   InRegs.ax = &H4E00
```

```
CALL INTERRUPTX(&H21, InRegs, OutRegs)

IF OutRegs.flags AND 1 THEN
 GetFirstMatchingFile$ = ""
 EXIT FUNCTION
END IF

DEF SEG = DTASegment%
MatchOffset% = DTAOffset% + 29
      :
```

Tutto ciò che resta ora è convertire la stringa ASCIIZ in una stringa BASIC e ritornare:

```
FUNCTION GetFirstMatchingFile$ (FileSpec$)

    DIM InRegs AS RegTypeX, OutRegs AS RegTypeX
    REM Ottiene l'indirizzo della Disk Transfer Area (DTA

    InRegs.ax = &H2F00
    CALL INTERRUPTX(&H21, InRegs, OutRegs)
    DTASegment% = OutRegs.es
    DTAOffset% = OutRegs.bx

    REM Ora si trova la prima corrispondenza

    NameFile$ = FileSpec$ + CHR$(0)
    InRegs.ds = VARSEG(NameFile$)
    InRegs.dx = SADD(NameFile$)
    InRegs.cx = 0
    InRegs.ax = &H4E00
    CALL INTERRUPTX(&H21, InRegs, OutRegs)

    IF OutRegs.flags AND 1 THEN
     GetFirstMatchingFile$ = ""
     EXIT FUNCTION
    END IF

    DEF SEG = DTASegment%
    MatchOffset% = DTAOffset% + 29

    Match$ = ""
    FOR i = 1 TO 13
     NewChar$ = CHR$(PEEK(MatchOffset% + i))
     IF NewChar$ = CHR$(0) THEN EXIT FOR
     Match$ = Match$ + NewChar$
    NEXT i
    DEF SEG

    GetFirstMatchingFile$ = Match$
```

E questo è tutto. Se si è trovata una corrispondenza, si restituisce il nome del file in una stringa BASIC. In caso contrario, si restituisce una stringa nulla. Inoltre, ora nella Disk Transfer Area si trovano le informazioni che servono a GetNextMatchingFile$( ) per continuare la ricerca.
Il listato 9.6 mostra la funzione GetFirstMatchingFile$( ) completa.

```
DECLARE FUNCTION GetFirstMatchingFile$ (FileSpec$)


TYPE RegTypeX
        ax      AS INTEGER
        bx      AS INTEGER
        cx      AS INTEGER
        dx      AS INTEGER
        bp      AS INTEGER
        si      AS INTEGER
        di      AS INTEGER
        flags   AS INTEGER
        ds      AS INTEGER
        es      AS INTEGER
END TYPE


DECLARE SUB INTERRUPTX (InNo AS INTEGER, InRgs AS RegTypeX,
                        OutRgs AS RegTypeX)

CLS

FileSpec$ = "*.DAT"

MatchFile$ = GetFirstMatchingFile$(FileSpec$)

IF MatchFile$ <> "" THEN
    PRINT "Il primo file che soddisfa i requisiti è: ",
          MatchFile$
ELSE
    PRINT "Non ci sono file corrispondenti."
END IF


END


FUNCTION GetFirstMatchingFile$ (FileSpec$)
```

continua

Listato 9.6. *La funzione GetFirstMatchingFile$*

```
    DIM InRegs AS RegTypeX, OutRegs AS RegTypeX

    REM Ottiene l'indirizzo della Disk Transfer Area (DTA

    InRegs.ax = &H2F00
    CALL INTERRUPTX(&H21, InRegs, OutRegs)
    DTASegment% = OutRegs.es
    DTAOffset% = OutRegs.bx

    REM Ora si trova la prima corrispondenza

    NameFile$ = FileSpec$ + CHR$(0)
    InRegs.ds = VARSEG(NameFile$)
    InRegs.dx = SADD(NameFile$)
    InRegs.cx = 0
    InRegs.ax = &H4E00
    CALL INTERRUPTX(&H21, InRegs, OutRegs)

    IF OutRegs.flags AND 1 THEN
     GetFirstMatchingFile$ = ""
     EXIT FUNCTION
    END IF

    DEF SEG = DTASegment%
    MatchOffset% = DTAOffset% + 29

    Match$ = ""
    FOR i = 1 TO 13
     NewChar$ = CHR$(PEEK(MatchOffset% + i))
     IF C$ = CHR$(0) THEN EXIT FOR
     Match$ = Match$ + NewChar$
    NEXT i

    DEF SEG

    GetFirstMatchingFile$ = Match$

END FUNCTION
```

Listato 9.6. *La funzione GetFirstMatchingFile$( )*

Ed ecco come utilizzare GetFirstMatchingFile$( ):

```
REM Esempio di uso di GetFirstMatchingFile$

DECLARE FUNCTION GetFirstMatchingFile$ (FileSpec$)

FileSpec$ = "*.DAT"
```

```
MatchFile$ = GetFirstMatchingFile$(FileSpec$)

IF MatchFile$ <> THEN
    PRINT "Il primo file che soddisfa i requisiti è ", MatchFile$
ELSE
    PRINT "Non ci sono corrispondenze."
END IF
```

Questo esempio restituisce il primo nome di file nella directory corrente con estensione .DAT. Bisogna ricordare che questa funzione restituisce solo il nome del *primo* file che soddisfa i requisiti. Per continuare la ricerca, bisogna usare GetNext-MatchingFile$( ). Inoltre, vengono restituiti solo file che abbiano attributo 0 (cioè non sono presi in considerazione eventuali file nascosti, di sistema, o di sottodirectory).

PER TROVARE I FILE SUCCESSIVI

Ora bisogna usare il successivo servizio del DOS, &H4F, per continuare la ricerca di file che soddisfino i requisiti, e si può scrivere una funzione GetNextMatchingFile$( ) che svolga questo lavoro. Questa funzione cercherà tutti gli altri file che soddisfano le specifiche, dopo che è stata usata GetFirstMatchingFile$( ). Semplicemente si continua a chiamare GetNextMatchingFile$( ) finché non restituisce una stringa nulla, " ": a quel punto non esistono più file che soddisfino i requisiti.
Questa funzione è molto simile a GetFirstMatchingFile$( ), salvo che usa il servizio &H4F dell'interrupt &H21 per cercare il *successivo* file che soddisfa le specifiche. Si comincia con il cercare come prima l'indirizzo della DTA, poi si chiama il servizio &H4F (non è necessaria alcuna immissione: tutto ciò che serve si trova già nella DTA):

```
FUNCTION GetNextMatchingFile$

    REM Ottiene l'indirizzo della Disk Transfer Area (DTA)
    InRegs.ax = &H2F00
    CALL INTERRUPTX(&H21, InRegs, OutRegs)
    DTASegment% = OutRegs.es
    DTAOffset% = OutRegs.bx

    REM Ora trova il successivo file che soddisfa i requisiti

    InRegs.ax = &H4F00
    CALL INTERRUPTX(&H21, InRegs, OutRegs)
        :
        :
```

Se, al ritorno da INTERRUPTX(), il flag di carry è impostato, vuol dire che non è stato trovato alcun file che soddisfi i requisiti, e lo si deve indicare restituendo una stringa nulla, "". In caso contrario, come nella funzione precedente si legge il nome del file dalla DTA:

```
FUNCTION GetNextMatchingFile$

   DIM InRegs AS RegTypeX, OutRegs AS RegTypeX
   REM Ottiene l'indirizzo della Disk Transfer Area (DTA)

   InRegs.ax = &H2F00
   CALL INTERRUPTX(&H21, InRegs, OutRegs)
   DTASegment% = OutRegs.es
   DTAOffset% = OutRegs.bx

   REM Ora trova il successivo file che soddisfa i requisiti

   InRegs.ax = &H4F00
   CALL INTERRUPTX(&H21, InRegs, OutRegs)

   IF OutRegs.flags AND 1 THEN
    GetNextMatchingFile$ = ""
    EXIT FUNCTION
   END IF

   DEF SEG = DTASegment%
   MatchOffset% = DTAOffset% + 29

   Match$ = ""
   FOR i = 1 TO 13
    NewChar$ = CHR$(PEEK(MatchOffset% + i))
    IF NewChar$ = CHR$(0) THEN EXIT FOR
    Match$ = Match$ + NewChar$
   NEXT i

   DEF SEG

   GetNextMatchingFile$ = Match$
```

E questo è tutto. Se c'è un file che corrisponde alla specifica, si restituisce il suo nome in una stringa BASIC; in caso contrario, si imposta GetNextMatchingFile$ a una stringa nulla e si esce.
Il listato 9.7 mostra la funzione completa.

```
DECLARE FUNCTION GetNextMatchingFile$ ()

TYPE RegTypeX
        ax      AS INTEGER
        bx      AS INTEGER
        cx      AS INTEGER
        dx      AS INTEGER
        bp      AS INTEGER
        si      AS INTEGER
        di      AS INTEGER
        flags   AS INTEGER
        ds      AS INTEGER
        es      AS INTEGER
END TYPE

DECLARE SUB INTERRUPTX (InNo AS INTEGER, InRgs AS RegTypeX,
                        OutRgs AS RegTypeX)

FUNCTION GetNextMatchingFile$

   DIM InRegs AS RegTypeX, OutRegs AS RegTypeX

   REM Ottiene l'indirizzo della Disk Transfer Area (DTA)

   InRegs.ax = &H2F00
   CALL INTERRUPTX(&H21, InRegs, OutRegs)
   DTASegment% = OutRegs.es
   DTAOffset% = OutRegs.bx

   REM Ora trova il successivo file che soddisfa i requisiti
   InRegs.ax = &H4F00
   CALL INTERRUPTX(&H21, InRegs, OutRegs)

   IF OutRegs.flags AND 1 THEN
    GetNextMatchingFile$ = ""
    EXIT FUNCTION
   END IF

   DEF SEG = DTASegment%
   MatchOffset% = DTAOffset% + 29
   Match$ = ""
   FOR i = 1 TO 13
    NewChar$ = CHR$(PEEK(MatchOffset% + i))
    IF NewChar$ = CHR$(0) THEN EXIT FOR
    Match$ = Match$ + NewChar$
   NEXT i
```

continua

Listato 9.7. *La funzione GetNextMatchingFile$( )*

```
    DEF SEG

    GetNextMatchingFile$ = Match$

END FUNCTION
```

Listato 9.7. *La funzione GetNextMatchingFile$( ).*

Ed ecco come utilizzare sia GetFirstMatchingFile$( ) sia GetNextMatchingFile$( ):

```
REM Esempio di uso di GetFirstMatchingFile$ e
GetNextMAtchingFile$

DECLARE FUNCTION GetFirstMatchingFile$ (FileSpec$)
DECLARE FUNCTION GetNextMatchingFile$ ( )

FileSpec$ = "*.DAT"

MatchFile$ = GetFirstMatchingFile$(FileSpec$)

IF MatchFile$ <> " " THEN
    PRINT "Il primo file che soddisfa i requisiti è: ",
MatchFile$
    MatchFile$ = GetNextMatchingFile$
    DO WHILE MatchFile$ <> ""
        PRINT "Il successivo file che soddisfa i requisiti è: ",
              MatchFile$
        MatchFile$ = GetNextMatchingFile$
    LOOP
ELSE
    PRINT "Non ci sono file che soddisfino i requisiti."
END IF
```

Come si può notare, prima si usa GetFirstMatchingFile$( ), si controlla il risultato e poi si procede con GetNextMatchingFile$( ). Se da GetFirstMatchingFile$( ) si ottiene una stringa nulla, non ci sono file che corrispondano alla specifica e si chiude il programma. Se invece è stata trovata una corrispondenza, la si stampa e si procede in ciclo su GetNextMatchingFile$( ) finché non restituisce "", al che si può uscire:

```
DO WHILE MatchFile$ <> ""
    PRINT "Il successivo file che soddisfa i requisiti è: ",
MatchFile$
    MatchFile$ = GetNextMatchingFile$
LOOP
```

Si è fatta molta strada, nella rassegna del BIOS e del DOS e si è anche visto al lavoro alcuni dei loro servizi. Ora è venuto il momento di completare il quadro.

IL RESTO DEL DOS

Gli ultimi servizi dell'interrupt &H21 sono un po' un'accozzaglia: sono stati aggiunti progressivamente con le nuove versioni del DOS. Eccoli:

Interrupt &H21	Servizi &H50-&H53	Interni al DOS
Interrupt &H21	Servizio &H54	Ottiene lo stato di verifica
Interrupt &H21	Servizio &H55	Interno al DOS
Interrupt &H21	Servizio &H56	Ridenominazione file
Interrupt &H21	Servizio &H57	Ottiene o imposta data e ora di un file
Interrupt &H21	Servizio &H58	Interno al DOS
Interrupt &H21	Servizio &H59	Ottiene un errore esteso (DOS 3+)
Interrupt &H21	Servizio &H5A	Creazione di file unico (DOS 3+)
Interrupt &H21	Servizio &H5B	Creazione di nuovo file (DOS 3+)
Interrupt &H21	Servizio &H5C	Blocca e sblocca l'accesso a un file (DOS 3+)
Interrupt &H21	Servizio &H5E00	Ottiene il nome della macchina (DOS 3+)
Interrupt &H21	Servizio &H5E02	Imposta la configurazione della stampante (DOS 3+)
Interrupt &H21	Servizio &H5E03	Ottiene la configurazione della stampante (DOS 3+)
Interrupt &H21	Servizio &H5E03	Ridirige dispositivo (DOS 3+)
Interrupt &H21	Servizio &H5E04	Cancella ridirezione (DOS 3+)
Interrupt &H21	Servizio &H61	Riservato
Interrupt &H21	Servizio &H62	Ottiene il prefisso del segmento di programma (DOS 3+)
Interrupt &H21	Servizio &H63-64	Riservati
Interrupt &H21	Servizio &H65	Ottiene informazioni estese sulla localizzazione
Interrupt &H21	Servizio &H6601	Ottiene la code page globale
Interrupt &H21	Servizio &H6602	Imposta la code page globale
Interrupt &H21	Servizio &H67	Imposta il conteggio degli handle (DOS 3.30)
Interrupt &H21	Servizio &H68	Commit file (buffer di scrittura) (DOS 3.30)
Interrupt &H21	Servizio &H6C	Apertura/creazione estesa (DOS 4.0)

***Nota**: Se il numero del servizio è lungo 16 bit (per esempio &H5F03) bisogna caricare nel registro ax quel numero per intero.*

Con questo si conclude la rassegna dei servizi dell'interrupt &H21: come si può notare, questo interrupt può essere molto utile per chi programma in BASIC.

***Nota**: Ulteriori informazioni su questo interrupt si possono trovare nell'Appendice.*

I restanti interrupt del DOS in genere non sono molto utili, invece, fatta eccezione per &H25 e &H26, che leggono e scrivono settori su disco (ulteriori informazioni si possono trovare nell'Appendice) e per &H27, che consente la creazione di programmi TSR (anche se in gran parte è stato superato dal servizio &H3C dell'interrupt &H21). Questi sono gli interrupt del DOS restanti:

Interrupt &H22	Termina indirizzo
Interrupt &H23	Indirizzo di uscita di Control-Pausa
Interrupt &H24	Gestore di errori critici
Interrupt &H25	Lettura assoluta da disco
Interrupt &H26	Scrittura assoluta su disco
Interrupt &H27	TSR
Interrupt &H28-&H2E	Interni al DOS
Interrupt &H2F	Interrupt multiplo
Interrupt &H&H30-&H3F	Riservati al DOS

Gli altri interrupt, fino a &HFF, non sono di grande aiuto, fatta eccezione per &H67, che offre il supporto per lo standard LIM 4.0 e la memoria espansa:

Interrupt &H40-&H5F	Riservati
Interrupt &H60-&H66	Riservati per il software d'utente
Interrupt &H67	Supporto per LIM 4.0
Interrupt &H68-&H7F	Non utilizzati
Interrupt &H80-&H85	Riservati per il BASIC
Interrupt &H86-&HF0	Utilizzati dall'interprete BASIC
Interrupt &HF1-&HFF	Non utilizzati

E questo è tutto per gli interrupt, da 0 a &HFF.

CONCLUSIONE

Abbiamo così completato la nostra rassegna del BIOS e del DOS, dall'interrupt 0 all'interrupt &HFF, e abbiamo anche messo all'opera alcuni dei servizi disponibili. Ora è tempo di dedicarsi davvero alla programmazione di basso livello e di compiere una digressione nel linguaggio assembly, con il capitolo 10.

CAPITOLO 10

IL LINGUAGGIO ASSEMBLY

In questo capitolo, si comincerà a lavorare con la macchina ai livelli più bassi: in questo ambito si possono aumentare enormemente la velocità e le capacità del BASIC. Il capitolo tratta gli elementi essenziali del linguaggio assembly; nel prossimo capitolo si vedrà come lo si possa interfacciare al BASIC. Si vedrà come sia possibile scrivere sottoprogrammi e funzioni BASIC interamente in linguaggio assembly, per ottenere grande potenza ed efficacia. La cosa più importante è partire da una buona base: è questo l'obiettivo del capitolo, in cui si impareranno gli elementi fondamentali della programmazione in linguaggio assembly. In effetti, attraverso l'uso della routine INTERRUPT( ), si è già fatta un po' di strada in questa direzione.

LINGUAGGIO MACCHINA

Perché un computer faccia qualcosa, bisogna fornirgli istruzioni in linguaggio macchina, byte comprensibili effettivamente solo al microprocessore. Spesso, solo parte dell'istruzione in linguaggio macchina viene utilizzata per dire al computer che cosa deve fare; il resto dell'istruzione è costituito da dati. Per esempio, si può scrivere un'istruzione per inserire il byte &HFF in una certa locazione di memoria. Parte dell'istruzione servirà a dire al microprocessore che si vuole memorizzare un numero, parte servirà a dire in quale locazione della memoria si vuole memorizzare il numero e parte sarà costituita dal numero stesso, &HFF.
Le istruzioni in linguaggio macchina posso essere costituite da molti byte, ma i codici dei dati e i codici di istruzione sono sempre costituiti da un numero intero di byte, mai da frazioni. Per esempio, certe istruzioni in linguaggio macchina possono essere costituite semplicemente da istruzioni per il microprocessore (figura 10.1).

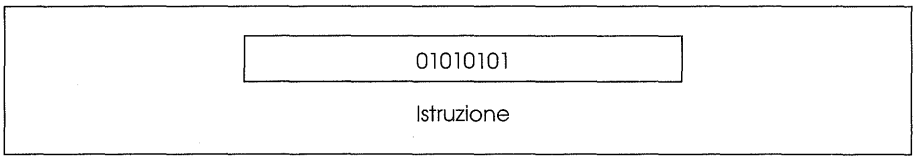

Figura 10.1

Altre invece sono un insieme di istruzione e dati (figura 10.2).

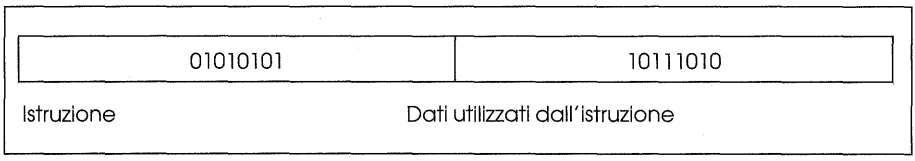

Figura 10.2

O addirittura sono costituite prevalentemente da dati (figura 10.3).

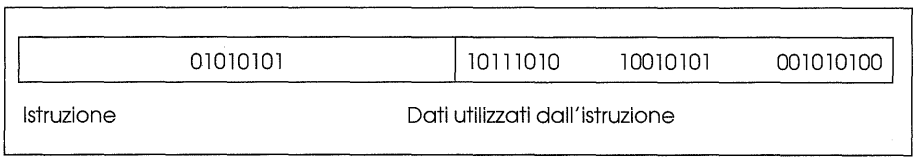

Figura 10.3

I dati utilizzati dall'istruzione sono o indirizzi di memoria o dati, come quel &HFF che prima si voleva memorizzare in una certa locazione di memoria.
Leggere il codice binario è estremamente difficile. Si pensi a una pagina di numeri, tutti 0 o 1: anche se le istruzioni fossero convertite in esadecimale, bisognerebbe andare a consultare il significato di ciascun byte prima di capire che cosa succede (nei manuali che accompagnano l'assembler ci sono di solito tabelle che elencano il significato delle istruzioni binarie).

LINGUAGGIO ASSEMBLY

Ed ecco dove entra in scena il linguaggio assembly: è un intermediario diretto fra linguaggi macchina e linguaggio ordinario. Per ogni istruzione in linguaggio macchina esiste una singola istruzione in linguaggio assembly. Invece di usare un byte come 10101010B (mettiamo una B alla fine dei numeri binari per evidenziarli: non esiste un tipo binario in BASIC, perciò non possiamo indicare questi numeri con una notazione del tipo &B10101010), si usa una mnemonica inglese, come MOV AX,5. Questa istruzione, MOV AX,5, può essere molto concisa, ma è pur sempre un bel miglioramento rispetto al codice macchina corrispondente, &HB8 &H00 &H05. Quello che fa quest'istruzione è dire alla macchina di spostare (la parte MOV di MOV AX,5) il valore 5 nel registro AX.
Il lavoro svolto da un assembler è semplice: prende il programma scritto in linguaggio assembly e lo converte, un'istruzione alla volta, direttamente in linguaggio macchina. Poi il linguaggio macchina viene eseguito dal microprocessore. Vediamo qualche esempio in linguaggio assembly. Si tratta di istruzioni che sono state assemblate (cioè convertite in linguaggio macchina): accanto a ogni istruzione in linguaggio assembly si vede perciò anche la corrispondente istruzione in linguaggio macchina (tutti i numeri sono in esadecimale):

MOV	DI,00B0	→	BF B0 00
MOV	COUNTER,00B0		C7 06 C3 01 B0 00
MOV	BX,0080		BB 80 00
CMP	INDEX,00		80 3F 00
JZ	0670		74 12
CMP	[SI],0D		80 3C 0D
JZ	0666		74 03
MOVSB			A4
JMP	065E		EB F8

L'ISTRUZIONE MOV

L'istruzione fondamentale del linguaggio assembly è MOV, l'istruzione che sposta i dati fra registri e memoria o da registro a registro. La sintassi è:

```
MOV destinazione, origine.
```

I dati vengono spostati dall'origine alla destinazione. Se si ha qualcosa in memoria e si vuole lavorare con quell'elemento, si può usare MOV. Ecco come funziona:

```
MOV   AX,0FFFFH
```

Questo esempio mette il numero 0FFFFH (65535) in AX. 0FFFFH è il numero più grande che i registri (sono tutti registri a 16 bit) possono contenere. Si osservi che,

in linguaggio assembly, la H per "esadecimale" va alla fine del numero (0FFFFH) e non all'inizio come in BASIC (&HFFFF). Inoltre, se un numero inizia con una lettera, come nel caso di FFFFH, bisogna mettergli davanti uno 0 (0FFFFH) per indicare all'assembler che si tratta effettivamente di un numero e non di un nome.
Si può prendere 0FFFFH nel registro AX e spostarlo nel registro DX:

```
MOV  DX,AX   (Sposta i dati da AX in DX)
```

E ora DX e AX contengono lo stesso valore. Si può lavorare anche un byte alla volta:

```
MOV  DL,AL   (Sposta i dati da AL in DL)
```

Si possono anche spostare dati dalla memoria nei registri. Si supponga di avere una locazione di memoria che contiene uno 0. Lo si può muovere, poniamo, in CX, in questo modo:

```
MOV  CX,[Locazione di memoria]
```

Oppure si può spostare nella locazione di memoria quello che si trova in DX, in questo modo:

```
MOV  [Locazione di memoria],DX
```

Tuttavia, non si possono spostare direttamente dati da una locazione di memoria a un'altra locazione di memoria. Questa è una delle particolarità dei microprocessori 80x86: i dati non possono passare direttamente, con una sola istruzione, da una locazione all'altra in memoria. Bisogna trasferire il dato dalla locazione di memoria 1 in AX, poi da AX alla locazione di memoria 2:

```
MOV  AX,[Locazione di memoria 1]
MOV  [Locazione di memoria 2],AX
```

Non si può invece scrivere:

```
MOV  [Locazione di memoria 2],[Locazione di memoria 1]
```

UN ESEMPIO IN LINGUAGGIO ASSEMBLY

È sempre tutto più facile, attraverso un esempio. In tutte le versioni del DOS si trova un eccellente programma che si chiama DEBUG, il quale ha la capacità di assemblare piccoli programmi che si scrivono al momento: nel programma è presente un mini-assemblatore.
Si utilizzera questo miniassembler per convertire in linguaggio macchina un'istruzione MOV AX,5 e poi eseguirla, per vedere come il valore memorizzato in AX cambi da 0 a 5. Si avvii il programma DEBUG. DEBUG presenta il suo prompt sotto forma di trattino:

```
C:\>DEBUG
-
```

Il comando R di DEBUG sta per Registro e consente di vedere i contenuti di tutti i registri degli 80x86. Si possono cogliere facilmente i registri AX, BX, CX e DX (si noti che tutti i numeri visualizzati in DEBUG sono in esadecimale, come è tipico di tutti i debugger in linguaggio assembly):

```
C:\>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 9AEC04020F    CALL    0F02:04EC
```

Oltre ai registri:

```
C:\>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000  DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 9AEC04020F    CALL    0F02:04EC
```

sono mostrati i valori dei flag interni dell'80x86 (questi flag sono otto, come si è visto nel Capitolo 9):

```
C:\>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 9AEC04020F    CALL    0F02:04EC
```

Si sono già incontrati i flag carry e zero; le notazioni NZ e NC dicono che, per il momento, non sono impostati. I flag sono utilizzati nei salti condizionali, con cui si lavorerà più avanti in questo capitolo, DEBUG dice inoltre la locazione di memoria corrente. Qui, ci si trova alla locazione 0EF1:0100:

```
C:\>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 9AEC04020F    CALL    0F02:04EC
```

L'ultima parte della videata indica che cosa si trova nella locazione di memoria corrente. In questo caso, si tratta dei byte che seguono l'indirizzo indicato nella videata di R:

```
C:\>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 9AEC04020F    CALL    0F02:04EC
```

DEBUG cerca di raggruppare i byte, a partire dalla locazione di memoria corrente (che contiene un solo byte), in modo da fornire un'istruzione in linguaggio macchina valida. Poi fornisce una traduzione in linguaggio assembly dell'istruzione in linguaggio macchina che inizia alla locazione corrente.
Quando non è presente un'istruzione linguaggio macchina (cosa che succede spesso), la traduzione è senza senso, come in questo caso. Poiché è stato appena avviato DEBUG, non c'è alcun programma da vedere. DEBUG ha preso semplicemente dei byte rimasti nella memoria del computer e cerca di trarne un senso. In effetti, la traduzione fornita da DEBUG non significa alcunché:

```
C:\>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 9AEC04020F       CALL    0F02:04EC
```

Utilizzeremo il comando A (che sta per Assemble) per inserire il programma d'esempio, costituito da una sola riga: MOV AX,5. Il comando A richiede l'indicazione dell'indirizzo dal quale cominciare a depositare in memoria le istruzioni in linguaggio macchina che genererà.
L'indirizzo corrente è 0EF1:0100, e si dirà a DEBUG di cominciare ad assemblare il linguaggio macchina in quella posizione, attraverso la notazione stenografica A100:

```
C:\>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 9AEC04020F    CALL    0F02:04EC
-A100                                Questo è il comando A100
03F1:0100                            La risposta di DEBUG.
```

> ***Consiglio:*** Il comando Assemble di DEBUG offre un metodo facile per verificare istruzioni in linguaggio macchina, facendo risparmiare il tempo necessario per modificare un file sorgente e per assemblarlo. Inoltre, se si devono scoprire gli effettivi byte di linguaggio macchina che corrispondono a qualche istruzione in linguaggio assembly, basta dare il comando Assemble e poi il comando di "memory dump", D.

Dopo il comando A100, DEBUG restituisce la riga 0EF1:0100, che indica l'indirizzo al quale verrà depositato il codice assemblato. Si scrive semplicemente MOV AX,5, seguito da un Invio:

```
C:\>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 9AEC04020F    CALL    0F02:04EC
-A100
03F1:0100 MOV    AX,5     Si digita "Mov AX,5<Invio>"
0EF1:0103
```

DEBUG poi rimane in attesa dell'istruzione successiva che si vorrà inserire all'indirizzo 0EF1:0103. Per ora non ci sono altre istruzioni da assemblare, perciò si dà a DEBUG un Invio. Per DEBUG la riga vuota significa che si è finito di assemblare, e ritorna al suo prompt normale.
Abbiamo assemblato così la nostra prima riga di linguaggio assembly. Per capire che cosa è successo, si ricordi che il comando R visualizza la locazione di memoria corrente e l'istruzione che lì inizia. Poiché MOV AX,5 è stata assemblata alla locazione di memoria corrente, si dia il comando R per dare un'occhiata:

```
C:\>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 9AEC04020F    CALL    0F02:04EC
-A100
03F1:0100 MOV    AX,5
0EF1:0103
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 B80500        MOV    AX,0005
```

Si può vedere l'istruzione (si notino i byte di linguaggio macchina che corrispondono a MOV AX,5 nella visualizzazione di DEBUG). Eseguire l'istruzione è semplice. DEBUG ha un comando di trace e battendo T una volta si eseguirà l'istruzione corrente e si passerà alla locazione di memoria successiva:

```
C:\>DEBUG
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 9AEC04020F    CALL    0F02:04EC
-A100
03F1:0100 MOV    AX,5
0EF1:0103
-R
AX=0000 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0100  NV UP EI PL NZ NA PO NC
0EF1:0100 B80500        MOV    AX,0005
-T              Questo comando farà eseguire l'istruzione MOV.
```

```
AX=0005 BX=0000 CX=0000 DX=0000 SP=FFEE BP=0000 SI=0000 DI=0000
DS=0EF1 ES=0EF1 SS=0EF1 CX=0EF1 IP=0103  NV UP EI PL NZ NA PO NC
0EF1:0103 020F          ADD    CL,[BX]
DS:0000=CD
```

Dopo il comando T, DEBUG visualizza i nuovi contenuti dei registri e i valori dei flag: AX contiene il valore 5. Tutti i flag sono rimasti immutati, mentre è cambiata la locazione di memoria, che ora è 0EF1:0103, non più 0EF1:0100, perché l'istruzione in linguaggio macchina corrispondente a MOV AX,5 occupa in memoria tre byte (0B8H 05H 00H). L'istruzione successiva, quindi, comincerà tre byte più avanti: perciò 100 è diventato 103.

IL PRIMO PROGRAMMA IN LINGUAGGIO ASSEMBLY

DEBUG consente non solo di assemblare programmi, ma anche di scriverli su disco. Si userà DEBUG per assemblare un primo programma in linguaggio assembly. Il programma si limiterà a scrivere la lettera Z, per poi uscire.
Si comincerà con il comando A. Come prima, il codice in linguaggio macchina verrà memorizzato a partire dalla locazione 0100H:

```
c:\>DEBUG
-A1000
0EF1:0100
```

Per stampare la lettera si utilizzerà il servizio 2 dell'interrupt 21H. Si digitino semplicemente le seguenti istruzioni in linguaggio assembly alla lettera, facendo seguire un <Invio> al prompt 03F1:0108 per concludere la fase di assemblaggio:

```
c:\>DEBUG
-A100
0EF1:0100 MOV AH,2
0EF1:0102 MOV DL,5A
0EF1:0104 INT 21
0EF1:0106 INT 20
0EF1:0108          Qui digitare semplicemente <Invio>.
-A100
```

Il programma semplicemente carica dei valori in AH e DL mediante le istruzioni MOV. Si chiama così il servizio 2 dell'interrupt 21H, e questo servizio stampa il codice ASCII che si trova in DL (dove è stato collocato il codice ASCII della Z, 5AH).
Poi si impartisce direttamente una istruzione di interrupt 21H (INT 21H) seguita da INT 20H.

Dal capitolo precedente si ricorderà che l'interrupt 20H è utilizzato per concludere i programmi a livello di linguaggio assembly e ritornare al prompt del DOS. Questo è il modo in cui si concluderanno tutti i nostri programmi. Si può dare al programma il nome STAMPAZ.COM, vista la sua funzione. Il nome si assegna in questo modo:

```
c:\>DEBUG
-A1000
0EF1:0100 MOV AH,2
0EF1:0102 MOV DL,5A
0EF1:0104 INT 21
0EF1:0106 INT 20
0EF1:0108
-NSTAMPAZ.COM
```

Ora si può scrivere il programma su disco (DEBUG lo scriverà nella directory corrente). DEBUG ha bisogno di sapere quanti byte deve scrivere: in questo caso, il programma va dalla locazione 0100H alla 0107H. Ogni locazione contiene un byte, perciò si tratta in tutto di 8 byte.

> ***Consiglio:*** DEBUG consente di effettuare operazioni aritmetiche in esadecimale, con il comando H, cioè Hex. per esempio, H 8 2 restituisce sia la somma sia la differenza di 8 e 2, cioè A e 6.

Il comando W (Write, scrivi) di DEBUG legge direttamente dal registro CX il numero dei byte da scrivere come file. Questo significa che, per scrivere gli 8 byte del programma STAMPAZ.COM, si deve caricare il valore 8 nel registro CX e poi impartire il comando W. Per spostare 8 in CX, si può usare di nuovo il comando R (Register). Se si usa il comando R senza argomenti, DEBUG presenta la sua visualizzazione standard.
Se si digita invece il comando RCX si dice a DEBUG che si vuole modificare il valore in CX (con le ovvie modifiche questo vale per qualsiasi registro). DEBUG visualizza il valore corrente in CX (che sarà 0000) e presenta un prompt a forma di punto, dopo il quale si digita il nuovo valore di CX, 8, e un Invio. Poi si può scrivere STAMPA.COM su disco impartendo il comando W:

```
c:\>DEBUG
-A1000
0EF1:0100 MOV AH,2
0EF1:0102 MOV DL,5A
0EF1:0104 INT 21
0EF1:0106 INT 20
0EF1:0108
-NSTAMPAZ.COM
-RCX
CX 0000
:8
```

```
-W                    Il comando W
Scrittura di 00008 byte in corso
-Q
```

(La Q finale è il comando per uscire da DEBUG: Q sta per Quit, uscita.) In questo modo si è scritto un programma applicativo di ben 8 byte (!). Si provi ad eseguirlo:

```
C:\>STAMPAZ
Z
C:\>
```

STAMPAZ fa esattamente quello che si voleva: stampa una Z ed esce.
Per la prima volta, si è raggiunto direttamente l'interrupt 21H (cioè senza l'intermediazione di INTERRUPT( )), e si è utilizzato il servizio 2 per stampare un carattere sullo schermo. Ecco alcuni altri servizi dell'interrupt 21H per l'I/O a caratteri, che si riveleranno utili nel corso del capitolo:

Servizio	**Nome**	**Imposta**	**Che cosa succede**
1	Immissione da tastiera	AH = 1	Il codice ASCII del tasto premuto viene restituito in AL
2	Emissione di carattere	AH = 2, DI=cod. ASCII	Viene visualizzato sullo schermo il carattere corrispondente al codice ASCII in DL.
9	Emissione di stringa	AH=9, DS:DX= indirizzo stringa di caratteri da stampare	Stampa una stringa du byte dalla memoria sullo schermo (questo servizio verrà utilizzato nel prossimo capitolo).

Armati di un minimo di esperienza, ora si può cominciare a scrivere qualche programma in linguaggio assembly senza utilizzare DEBUG. Bisogna però prima vedere come i chip dell'80x86 accedono alla memoria. Nel linguaggio assembly l'uso della memoria è molto più importante che in BASIC. In effetti, sapere come ci si deve muovere a questo proposito è essenziale.

SEGMENTAZIONE DELLA MEMORIA

L'indirizzo 0EF1:0100 è costituito da due numeri esadecimali, ciascuno di 16 bit. Come si è visto nel Capitolo 5, è una notazione normale per gli indirizzi: per avere un indirizzo completo sono necessarie due word. La parte 0EF1H in 0DF1:0100 è l'indirizzo del segmento di quella particolare locazione di memoria, mentre 0100H è l'indirizzo dell'offset (figura 10.4)

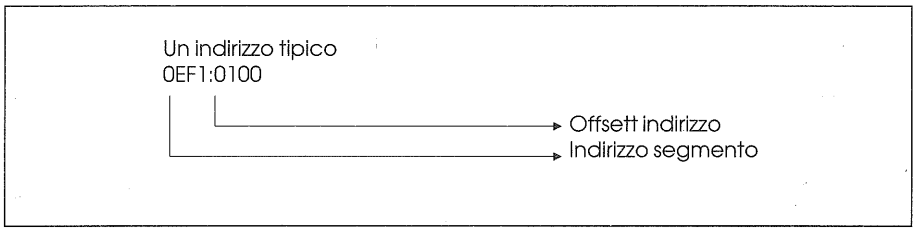

Figura 10.4

Sotto DOS, si può accedere a un massimo di un megabyte di memoria, cioè 2^20 byte. Per specificare così tante locazioni di memoria, sono necessari numeri lunghi 20 bit, ma il numero massimo di bit che può essere memorizzato nei registri dell'80x86 (fatta eccezione per le macchine 386 e 486) è di 16 bit. Per costruire indirizzi a 20 bit con numeri a 16 bit, servono due numeri, l'indirizzo del segmento e l'indirizzo dello scostamento. Ecco come funziona: si sposta l'indirizzo di segmento a sinistra di un posto esadecimale e lo si somma all'indirizzo dello scostamento per ottenere un indirizzo reale a 20 bit (figura 10.5).

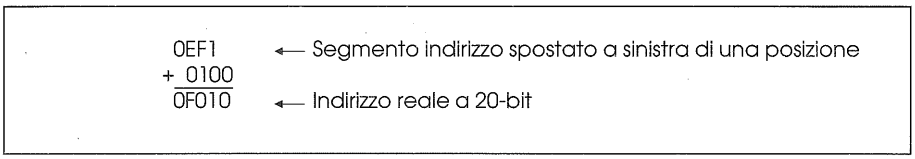

Figura 10.5

In altre parole, il byte 0EF1:0100 è in effetti il byte 0F010H, cioè il byte 61456, in memoria. L'indirizzo più basso è 0000:0000, il byte 0 in memoria, e l'indirizzo più alto è F000:FFFF, che corrisponde al byte OFFFFFH, cioè 1048575. Con questi indirizzi a 20 bit si può fare riferimento a 1 megabyte, 1024 K (figura 10.6).

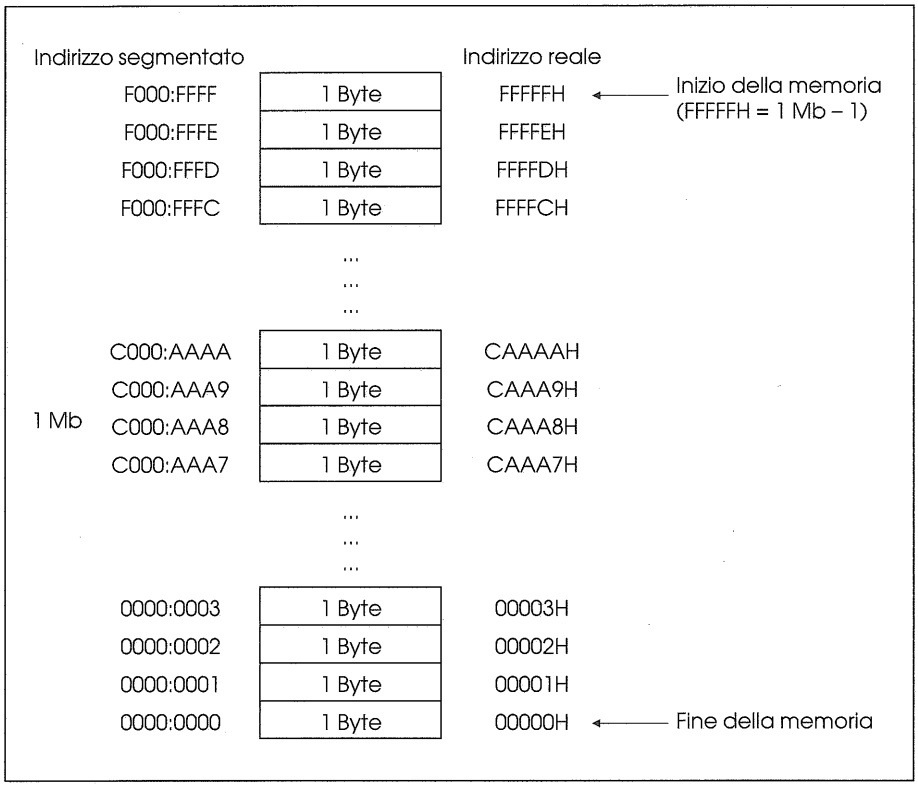

Figura 10.6

SEGMENTI IN MEMORIA

Un segmento è un'area di memoria che può essere indirizzata con un particolare indirizzo di segmento; un segmento può andare da *xxxx*:0000 a *xxxx*:FFFF, 64K. Per esempio, il segmento che inizia al fondo della memoria, il segmento 0000, può estendersi da 0000:0000 a 0000:FFFF (mantenendo inalterato l'indirizzo di segmento, 0000). Una volta scelto un indirizzo di segmento, come 0000, si ha a disposizione uno spazio di lavoro di 64K che si può utilizzare senza dover cambiare nuovamente indirizzo di segmento.

Invece, anche se possono descrivere un'area tanto grande, i segmenti possono sovrapporsi. Il successivo possibile segmento dopo il segmento 0000 è il segmento 0001, che può estendersi da 0001:0000 a 0001:FFFF. Convertendo questi numeri in indirizzi a 20 bit si ottengono 00010H e 1000FH.

Il segmento 0001 inizia solo 16 byte (chiamato *paragrafo*) dopo il segmento 0000. Il segmento 0002 inizia solo 16 byte dopo il segmento 0001 e così via. La scelta di un

segmento fornisce uno spazio di lavoro di 64K, ma quei 64K si sovrappongono a molti altri segmenti (figura 10.7).

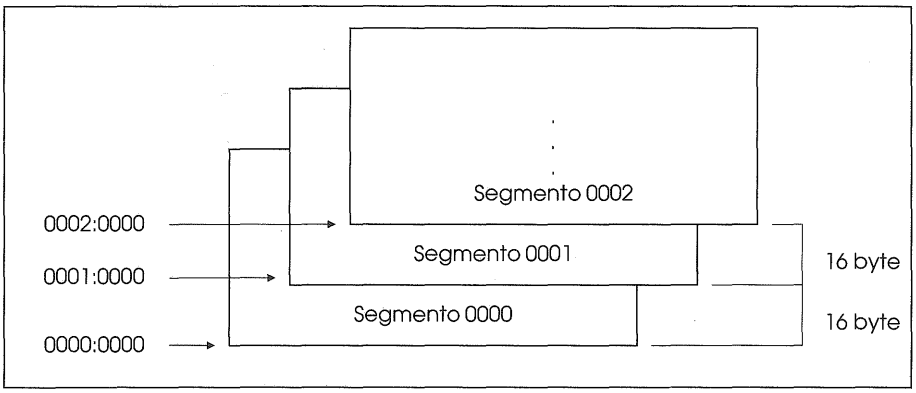

Figura 10.7

Per consentire la scelta del segmento desiderato come area di lavoro, l'80x86 fornisce quattro *registri di segmento* (già incontrati nel capitolo 9). Questi registri di segmento si impostano tipicamente all'inizio di un programma, oppure vengono impostati automaticamente. Bisogna ricordare, però, che in modo automatico definiscono semplicemente un'area di 64K. Se si vuole qualcosa al di fuori di quell'area, sta al programmatore impostarli secondo le sue esigenze.

I quattro registri di segmento sono CS, DS, ES e SS (le sigle stanno rispettivamente per Code Segment, Data Segment, Extra Segment e Stack Segment, cioè segmento del codice, segmento dei dati, segmento extra e segmento dello stack).

Registro di segmento	Significato	Usato con
CS	Code Segment	Le istruzioni del programma
DS	Data Segment	I dati su cui si vuol lavorare
ES	Extra Segment	Registro di segmento dati ausiliario
SS	Stack Segment	Impostato dal DOS, contiene lo stack.

Il segmento del codice è quello in cui vengono memorizzate le istruzioni del programma. Quando il programma viene caricato, il program loader sceglie il segmento del codice. Per gli esempi di questo libro non è necessario impostare il registro di segmento CS. (Vedi figura 10.8.)

Il registro DS contiene il valore del segmento dei dati. Questo registro è stato utilizzato nel Capitolo 9. Tutto quello che si vuole memorizzare come dato e che non deve essere eseguito dal calcolatore (celle in un foglio elettronico, per esempio, o testo in un word processor) può essere memorizzato qui.

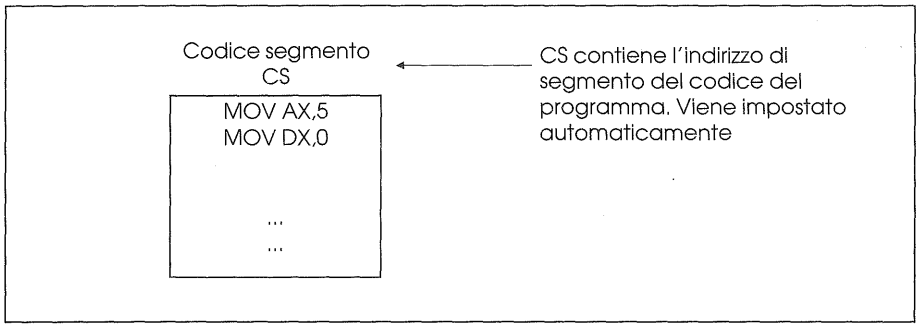

Figura 10.8

Di solito (se lo si deve fare) si imposta DS, il registro del segmento dei dati, all'inizio del programma e poi non lo si tocca più. Se però si vogliono leggere byte da locazioni lontane in memoria (per esaminare per esempio il buffer dello schermo o quello della tastiera), bisogna impostare DS prima di poterle indirizzare (come si usa DEF SEG in BASIC). Utilizzando DS come parola più significativa degli indirizzi, si può raggiungere, leggere o scrivere qualsiasi byte in memoria.
Supponiamo che il codice del programma sia nel segmento 2000H e che i dati siano nel segmento che inizia a 3000H (figura 10.9).

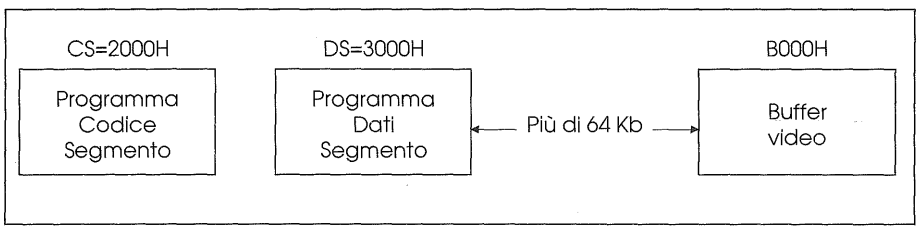

Figura 10.9

Ora supponiamo che si vogliano modificare i dati nel buffer del video (cioè le lettere che appaiono sullo schermo), che si trova, per la maggior parte dei monitor, al segmento B800H. Bisogna modificare il segmanto di dati che si sta usando, inserendo, in DS, l'indirizzo B800H (figura 10.10).
Poi si può fare riferimento, nelle istruzioni, a qualsiasi dato si trovi in quel segmento. Ogniqualvolta si leggono dati dalla memoria, l'80x86 controlla il valore di DS per la parte segmento dell'indirizzo. Le istruzioni che fanno riferimento a locazioni di memoria (come la MOV che si è usata prima) utilizzano automaticamente DS come indirizzo di segmento.

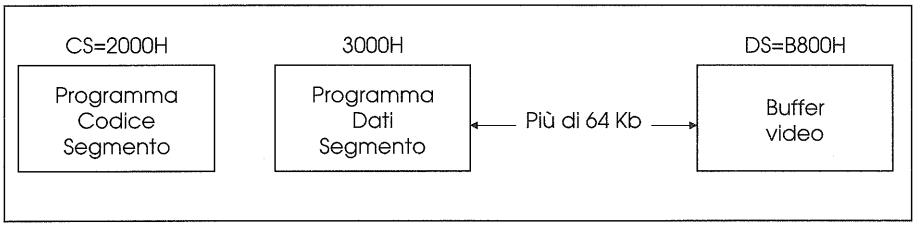

Figura 10.10

Il segmento extra può essere utilizzato come ulteriore segmento di dati, ma qui non se ne farà granché uso. C'è poi il segmento dello stack. Il DOS memorizza l'indirizzo di ritorno per le chiamate a funzioni e sottoprogrammi sullo *stack*, una parte speciale di memoria. Il BASIC passa invece i parametri sullo stack. Vedremo meglio il funzionamento dello stack, in questo capitolo e nel prossimo.

> ***Consiglio:*** Si possono usare sia il segmento extra sia il segmento dei dati quando si vogliono specificare sia un'origine sia una destinazione per i data, come, per esempio, nelle operazioni di copia.

I REGISTRI DI SEGMENTO AL LAVORO: I FILE .COM

I primi programmi che scriveremo saranno file .COM. Per default i file .COM tutti e quattro i registri di segmento sono impostati allo stesso valore, il segmento del codice. Qui lo si farà espressamente per non doversi preoccupare dei registri di segmento mentre si muovono faticosamente i primi passi nella programmazione in assembly.

Tutto quello che succede in un file .COM accade entro i limiti di uno spazio di lavoro di 64K e, quando il programma viene caricato, il DOS imposta automaticamente l'indirizzo di quel segmento. Questo significa in pratica che, per i file .COM, non si deve prestare attenzione all'impostazione dei registri di segmento. Qui, CS = DS = ES = SS e il DOS li imposta per noi.

Un file .COM è il tipo più semplice di programma funzionante che si può scrivere per un PC o un PS/2 (anche se il formato .COM non è più supportato sotto OS/2). Questo tipo di file è costituito solo da istruzioni in linguaggio macchina, pronte per l'esecuzione. La creazione di qualcuno di questi file ci permetterà di fare quel tanto di esperienza che serve per collegare il linguaggio assembly al BASIC.

DIRETTIVE PER L'ASSEMBLER

Quando si scrive un file sorgente in linguaggio assembly (che si identifica con l'estensione .ASM) per generare un file .COM, si deve specificare dove si vuole che venga collocato il codice, nel segmento del codice, se si vuole un segmento di dati distinto e altre cose. L'impostazione dei segmenti avviene attraverso *direttive per l'assembler*. Queste direttive non generano alcun codice, ma forniscono indicazioni all'assemblatore (e nient'altro). Il loro lavoro è simile a quello delle istruzioni SUB o FUNCTION in BASIC.
Quando si scrive un programma si possono impostare segmenti di codice o di dati, e per farlo si usano direttive per l'assembler. Se si imposta un segmento di codice, vi verranno memorizzate le stesse istruzioni del programma; se si imposta un segmento di dati, si predefiniscono variabili o costanti che il programma utilizzerà in seguito, oppure semplicemente si mette da parte dello spazio libero che verrà utilizzato dal programma. L'assembler si prenderà cura che queste informazioni vengano caricate nei segmenti giusti in memoria, se li si specifica con direttive di segmento.

LA DIRETTIVA .CODE

Tutto quello che si inserisce nei file .ASM sarà racchiuso entro una definizione di segmento. Si può aggiungere la definizione del segmento del codice al codice sorgente del programma STAMPAZ scritto in precedenza:

```
.CODE

        MOV     AH,2
        MOV     DL,5AH
        INT     21H
        INT     20H
        :
```

Da questo punto in poi, tutto quello che si scriverà verrà messo nel segmento del codice fino a che non finisce il file, o fino a che l'assembler non trova un'altra direttiva di segmento, per esempio .DATA, che inizierebbe un segmento di dati (come vedremo nel seguito del capitolo). Dato però che nei file .COM esiste un solo segmento, tutto va nel segmento del codice e questa è l'unica direttiva di segmento che ci serve.

ETICHETTE

I dati possono essere etichettati byte per byte, o word per word; anche le etichette sono direttive. Come in BASIC, si può etichettare un'istruzione nel programma stesso in modo da poter saltare da un'altra istruzione a quella etichettata, che può trovarsi anche a una certa distanza.
Per esempio, quando viene tradotta in linguaggio macchina, l'istruzione MOV AH,2 occupa 3 byte. Si può dare un'etichetta a questa istruzione, poniamo Start, e si può dare un'etichetta anche all'ultima istruzione (INT 20H), che potrebbe essere Exit:

```
.CODE

Start:  MOV     AH,2
        MOV     DL,5AH
        INT     21H
Exit:   INT     20H
```

Come in BASIC, per etichettare un'istruzione basta dare un nome seguito da un segno di due punti. Se, nel corso del programma si volesse uscire velocemente, si potrebbe semplicemente saltare all'etichetta Exit, e verrebbe eseguita l'istruzione INT 20H, che provoca l'uscita dal programma. Se si decidesse in tal senso, l'assembler dovrebbe sapere l'indirizzo dell'istruzione Exit, e lo trova contando il numero di byte in linguaggio macchina che ha prodotto dall'inizio del segmento del codice (che è stato etichettato con Start).

Consiglio: Bisogna fare attenzione, in linguaggio assembly, alle etichette molto lunghe, perché l'assembler legge solo i primi 31 caratteri: affinché due etichette siano considerate come diverse, devono avere qualche differenza nei primi 31 caratteri.

POSIZIONAMENTO DEL CODICE NEL SEGMENTO DEL CODICE

Quando vengono caricati, i file .EXE vengono posti all'inizio del segmento di codice loro assegnato, a CS:0000. La prima istruzione può iniziare esattamente in quella posizione. I file .COM, invece, sono dotati di una intestazione che viene caricata per prima, perciò a CS:0000 viene posta l'intestazione, non la prima istruzione del file .COM. Questa intestazione è lunga 100H (256 in decimale) byte, e va perciò da CS:0000 a CS:00FF e il file .COM, caricato di seguito in memoria, inizia esattamente a CS:0100 (figura 10.11).
Questo significa che il codice di STAMPA.ASM dovrà cominciare all'offset 0100H nel segmento del codice. È questo il motivo per cui nell'esempio di uso di DEBUG si è

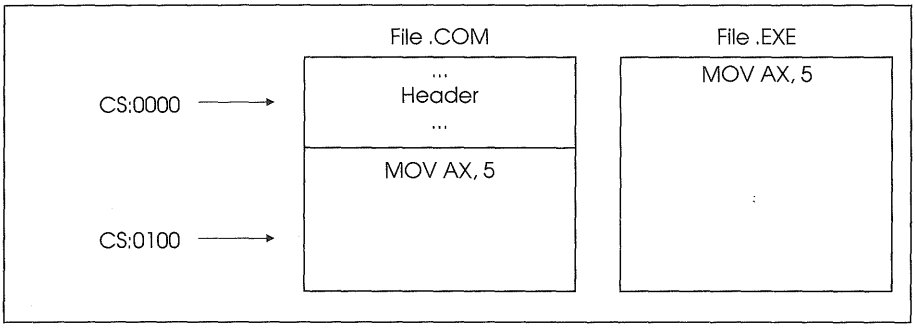

Figura 10.11

iniziato ad assemblare all'indirizzo 0100H (utilizzando A100). In un file .COM, le istruzioni in linguaggio macchina devono iniziare, all'offset 0100H.
Si può definire la posizione all'interno del segmento del codice con la direttiva ORG (che sta per origine):

```
.CODE
        ORG     100H
Start:  MOV     AH,2
        MOV     DL,5AH
        INT     21H
Exit:   INT     20H
```

Questo dice all'assembler di porre a 0100H l'offset di Start (la riga che segue immediatamente la direttiva ORG). L'assembler ora tratta l'istruzione MOV AH,2 come se fosse collocata in CS:0100 e non in CS:0000 (e tutto quello che segue viene trattato come se fosse collocato dopo questa istruzione). In questo modo, si è lasciato lo spazio giusto per l'intestazione, creata automaticamente per il file .COM.

LA DIRETTIVA END

L'ultima cosa necessaria è un *punto di ingresso* per il programma. In BASIC, il punto di ingresso viene fissato quando si definisce la procedura principale: il controllo viene sempre passato a quel punto per primo. In linguaggio assembly, il punto di ingresso può essere fissato in qualunque posizione del programma con la direttiva END. Ogni file .ASM deve concludersi con END, perché l'assembler possa sapere quando fermarsi. Contemporaneamente, si può fissare il punto di ingresso indicandone l'etichetta dopo la parola chiave END.
Nel nostro caso, si vuole che il punto di ingresso sia a 0100H nel segmento del codice. Questa istruzione è già stata etichettata Start, perciò la direttiva conclusiva sarà END Start. Ed è tutto. Il programma STAMPAZ.ASM, completo, è riportato nel listato 10.1.

```
CODE
        ORG     100H
Start:  MOV     AH,2
        MOV     DL,5AH
        INT     21H
Exit:   INT     20H
        END     Start
```

Listato 10.1. *Il programma STAMPAZ.ASM.*

ASSEMBLAGGIO DI STAMPAZ.ASM

Si può utilizzare un word processor o un editor per scrivere il programma in un file con il nome STAMPAZ.ASM, dopodiché si è pronti per usare l'assembler. Se si ha il Microsoft assembler (qui si userà il Microsoft MASM versione 5.1), bisogna battere il comando:

```
R:\>MASM STAMPAZ;
```

E il macro assembler dirà:

```
R:\>MASM STAMPAZ;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988.  All rights reserved.

  50144 + 31277 Bytes symbol space free

      0 Warning Errors
      0 Sever Errors
```

Il programma è stato assemblato, ma tutto quello che si è ottenuto è un file .OBJ. Il passo successivo consiste nell'eliminare alcune informazioni che l'assembler ha lasciato nel file .OBJ. Il linker viene utilizzato normalmente per combinare file .OBJ in grandi file eseguibili, ma anche un singolo file .OBJ deve passare attraverso il linker per poter diventare un file .COM.
Il linker, tra l'altro, controlla tutti i segmenti; dato che questo deve diventare un file .COM, esiste un unico segmento, il segmento del codice. I programmi che non sono di tipo .COM debbono avere una dichiarazione esplicita di tutti i segmenti, e qui il linker ci avvertirà che non abbiamo un segmento dello stack. Questo è il normale avvertimento che si riceve quando si producono file .COM.
LINK va usato in questo modo:

```
R:\>LINK STAMPAZ;
Microsoft (R) Overlay Linker Version 3.64
Copyright (C) Microsoft Corp 1983-1988. All rights reserved.

LINK : warning L4021: no stack segment
```

L'avvertimento è lì, ma il lavoro è ormai quasi completato: il linker ha preso il file .OBJ, STAMPAZ.OBJ, e ha prodotto un file .EXE, STAMPAZ.EXE. Non si voleva, però, un file .EXE, ma un file .COM. Per questo passo finale, che consiste nell'eliminare l'intestazione che il linker ha lasciato nel file .EXE, si esegue un programma DOS che si chiama EXE2BIN. Far intervenire questo programma sull'uscita fornita da LINK è l'ultimo passo del procedimento, e trasforma il file .EXE in formato .COM:

```
R:\>EXE2BIN STAMPAZ STAMPAZ.COM
```

E finalmente ecco il file .COM, pronto per partire. Si lanci STAMPAZ.COM: fa effettivamente quello che doveva fare, cioè visualizzare una Z, come nella versione di DEBUG. Abbiamo fatto un altro passo avanti: abbiamo un file .ASM che funziona.

AGGIUNGERE DATI AI PROGRAMMI IN ASSEMBLY

Questa però è solo metà della storia. Così com'è, STAMPAZ.ASM è un programma funzionante, ma molto primitivo. In quasi tutti i file .COM, si trovano anche dei dati.

Ogni variabile utilizzata in un programma fa parte dei dati. Si vuole leggere un file da disco e salvarne i contenuti nell'area dei dati, perché siano trattati come dati; oppure si vuole che nell'area dei dati siano memorizzati dei messaggi di programma come "Salve, gente".

Anche in assembly, come in BASIC, si possono usare variabili: si utilizzeranno le direttive DB, *define byte*, e DW, *define word*, per impostarle.

LE DIRETTIVE DB E DW

Si usa la direttiva DB per mettere da parte dei byte in memoria, per potervi memorizzare dei dati. Per definire un byte con il nome, poniamo, VALORE, si usa DB in questo modo:

```
VALORE DB 5
```

Se si vuole inserire questo dato in un segmento di dati, il codice può avere questo aspetto:

```
.CODE
MOV     AL,VALORE
        :

.DATA
VALORE DB 5
        :
```

Si è concluso il segmento del codice e iniziato il segmento dei dati utilizzando la direttiva .DATA. Non si può usare .DATA nei file .COM, ma solo nei file .EXE. Tutto quello che segue .DATA andrà nel segmento dei dati. In memoria, i segmenti del codice e dei dati possono essere organizzati come in figura 10.12.

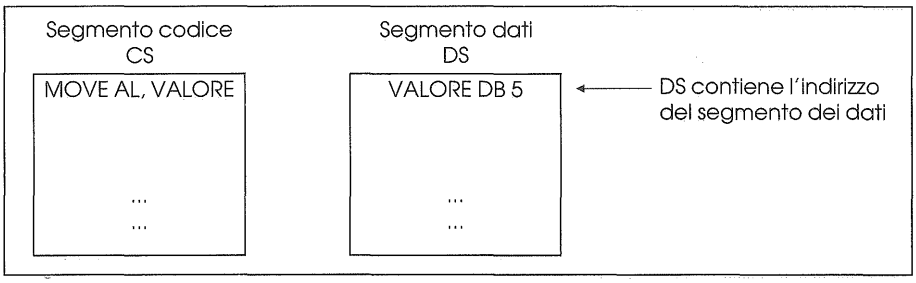

Figura 10.12

Ora si è liberi di leggere nel programma il dato contenuto in VALORE:

```
MOV AL,VALORE
```

Così si usa la memoria nel PC e nel PS/2: si mettono da parte delle locazioni con DB e si danno ai nomi di variabile tutti i byte messi da parte. Poi, quei nomi possono essere utilizzati esattamente come le variabili in BASIC.
Si possono memorizzare i dati anche una word alla volta con la direttiva DW, in questo modo:

```
.CODE
MOV     AX,VALORE_WORD
        :

.DATA
VALORE_WORD     DW &H1234
        :
```

In questo caso, si può lavorare con VALORE_WORD in questo modo:

```
MOV     AX,VALORE_WORD
```

Vediamo come funziona tutto questo aggiungendo dei dati a STAMPAZ.ASM. In questo caso, bisognerà metterli nel segmento del codice perché è disponibile un solo segmento. L'unico dato possibile è il carattere che si vuole stampare, Z:

```
CODE
        ORG     100H
        Carattere     DB "Z"
Start:  MOV     AH,2
        MOV     DL,5AH
        INT     21H
Exit:   INT     20H

        END     Start
```

DB dice all'assembler che i dati che seguono devono essere collocati nel programma senza interpretazione: sono dati. Bisogna predisporre una locazione (un byte), che è stata chiamata Carattere, che poi viene inizializzata con l'inserimento del carattere Z. L'assembler tradure la Z nel codice ASCII che serve alla macchina: 5AH. (In alternativa si sarebbe potuto definire Carattere DB 5AH: per l'assembler non fa differenza.) Ecco alcuni esempi di uso di DB:

```
Flag7    DB 0
Car_Z    DB "Z"
Numeri   DB 1,2,3,4,5,0          Vengono predisposti 6 byte
Prompt   DB "Da quanto tempo non telefoni più "
         DB "a tua madre?"
```

Quando si fa riferimento ai nomi Flag7, Car_Z, Numeri o Prompt, in effetti si fa riferimento al primo byte che segue DB. per esempio, se si scrivesse:

```
MOV     AH,Numeri
```

verrebbe caricato in AH l'1, cioè il primo numero dopo DB.
In STAMPAZ.ASM, ecco come si carica il carattere in DL, subito prima di stamparlo:

```
CODE

        ORG     100H
        Carattere     DB "Z"
Start:  MOV     AH,2
        MOV     DL,Carattere
        INT     21H
Exit:   INT     20H

        END     Start
```

In questo modo si è riusciti a etichettare e utilizzare una locazione di memoria. Tuttavia rimane un problema. Si presume che l'etichetta Start si trovi a 100H nel segmento del codice, ma ora che si è aggiunto 1 byte di memoria prima di essa, si troverà nella locazione sbagliata, cioè 101H. Per risolvere il problema, si farà qui

quello che fanno la maggior parte dei file .COM: si riserva un'area di dati all'inizio del programma e si aggiunge una *istruzione di salto*, in modo che, a 100H, la prima cosa che il microprocessore fa è saltare l'area dei dati fino alla prima istruzione. Il programma è presentato nel listato 10.2.

```
CODE

        ORG     100H
Start:  JMP     StampaZ
        Carattere     DB "Z"
StampaZ:MOV     AH,2
        MOV     DL,Carattere
        INT     21H
Exit:   INT     20H

        END     Start
```

Listato 10.2 *Programma PRINTZ.ASM*

Si è spostata l'etichetta Start in modo che punti a una nuova istruzione che si trova a 100H, ma quell'istruzione dice: JMP StampaZ. Questa istruzione consente di far passare il controllo, oltre l'area dei dati, all'etichetta StampaZ, per poi continuare; JMP è un'istruzione di linguaggio assembly esattamente analoga alla GOTO in BASIC. Per usare JMP, basta fornirle l'etichetta a cui deve saltare, come si è fatto qui (JMP StampaZ).
È tutto: STAMPAZ.ASM, aggiornato in modo da contenere dei dati, è finito. È pronto per essere assemblato ed eseguito. Se si vogliono inserire altri dati, è possibile indicarli dopo Carattere.
La figura 10.13 mostra come è organizzato in generale un file .COM.
C'è un'area predisposta per i dati (mediante DB o DW) e una parte per il codice del programma.

STRINGHE IN MEMORIA

Ancora non si è visto come si debbano memorizzare stringhe di caratteri. Una stringa di caratteri (come una STRING in BASIC) è semplicemente una serie di caratteri disposti uno dopo l'altro, che hanno un senso per noi ma non per il computer. Si vuole che vengano trattati insieme, ma il computer vede solo un gruppo di byte privi di alcuna relazione evidente.
Come in assembly anche in BASIC le stringhe sono memorizzate semplicemente come byte e hanno un terminatore che indica la fine della stringa. Anche se di solito

```
.CODE

        ORG     100H
Start:  JMP     PROG
        ____________________
                Questa è l'area dei dati. Qui si usa DB
        ____________________
PROG:
        ____________________
                E questa è la zona in cui va il programma
        ____________________
Exit:   INT     20H

        END     Start
```

Figura 10.13

le stringhe finiscono con 0 (in quello che si è chiamato formato ASCIIZ), non sempre è così. Se si vuole che la stringa termini con un byte 0, bisogna indicarlo esplicitamente:

```
Prompt   DB "Da quanto tempo non telefoni più "
         DB "a tua madre?", 0
```

L'assembler permette di memorizzare stringhe in questo modo, con l'uso delle virgolette come scorciatoia (altrimenti si dovrebbe usare DB per ciascuna lettera).

IL SERVIZIO 9 DELL'INT 21H: STAMPA UNA STRINGA

Il servizio per la stampa di stringhe, cioè il servizio 9 dell'interrupt 21H, è l'equivalente di PRINT per il linguaggio assembly. Per terminare la stringa per questo servizio, come ultimo carattere si deve aggiungere un segno di dollaro ($) e non un byte 0. Il segno $ indica al servizio 9 di smettere di stampare. Ecco come diventa il programma STAMPXYZ:

```
CODE

        ORG     100H
Start:  JMP     StampaZ
        Caratteri    DB "XYZ$"
```

```
StampaZ:MOV     AH,2
        MOV     DL,Caratteri
        INT     21H
Exit:   INT     20H

        END     Start
```

Si noti il carattere finale $ che corrisponte all'ultimo byte della stringa. È ancora necessario indicare al servizio 9 dove ritrovare in memoria questa stringa e modificare la chiamata dal servizio 2 al servizio 9. Questo servizio richiede l'indirizzo della stringa in DS:DX e se questo indirizzo, per esempio, corrisponde 0EF1:0105, sarà necessario caricare 0EF1H in DS e 0105H in DX.
Poiché si sta lavorando su un file .COM, il valore di DS non verrà mai modificato perciò DS è già impostato sul servizio 9. Quando il programma verrà avviato, DS punterà al segmento di codice (ovvero all'unico segmento esistente). Per assegnare a DX l'indirizzo dell'offset *Caratteri*, sarà necessario utilizzare la direttiva OFFSET come sotto riportato:

```
CODE

        ORG     100H
Start:  JMP     StampaZ
        Caratteri    DB "XYZ$"
StampaZ:MOV     AH,9
        MOV     DX,OFFSET Caratteri
        INT     21H
Exit:   INT     20H

        END     Start
```

La direttiva OFFSET fornisce il valore di scostamento di un'etichetta rispetto all'inizio del segmento dei dati (che nel caso attuale coincide con il segmento del codice). Si può pensare che restituisca un puntatore a quell'oggetto come VARPTR( ). Per esempio, la riga

```
MOV     DX, OFFSET Caratteri
```

caricherà in DX l'offset del primo byte di *Caratteri*. OFFSET è una direttiva molto comoda che si userà spesso (in particolare nel prossimo capitolo), poiché molti servizi di interrupt vogliono ricevere l'indirizzo dei dati.
A questo punto abbiamo un file .ASM funzionante che stampa XYZ invece di una semplice Z.

L'USO DEI COMMENTI

Prima di lasciare STAMPXYZ, due parole sui commenti. In linguaggio assembly si possono aggiungere commenti facendoli precedere da un punto e virgola (;), come si vede nel listato 10.3.

```
CODE

        ORG     100H                ;Impostazione per un file .COM
Start:  JMP     StampaZ             ;Salta oltre l'area dei dati
        Caratteri     DB "XYZ$"     ;Questa è la stringa da stampare
StampaZ:MOV     AH,9                ;Richiede servizio 9 dell'INT 21H
        MOV     DX,OFFSET Caratteri ;Punta alla nostra stringa
        INT     21H                 ;E qui la stampa
Exit:   INT     20H                 ;Fine del programma

        END     Start               ;Stabilisce che Start è
                                     il punto di ingresso
```

Listato 10.3 *STAMPXYZ.ASM: stampa "XYZ"*

Leggendo i commenti, si può vedere quale è il significato di ciascuna riga. In linguaggio assembly i commenti sono spesso più importanti ancora che in BASIC, e possono essere di grande aiuto.
Poste le fondamenta della scrittura di programmi, verrà ampliato il repertorio di istruzioni cominciando a leggere qualcosa dalla tastiera. Lavorare con l'immissione da tastiera fornirà un po' di esperienza in uno degli aspetti centrali per il linguaggio assembly: i salti condizionati (gli equivalenti di IF...THEN per il linguaggio assembly).

ACCETTARE INPUT DA TASTIERA

Il più fondamentale fra i servizi dell'interrupt 21H è il numero 1, che legge l'immissione da tastiera e svolge, nel linguaggio assembly, la funzione di INKEY$ (solo che aspetta l'immissione da tastiera). Quando viene premuto un tasto, l'eco viene inviato allo schermo e il relativo codice ASCII viene inviato al registro AL.

IL PROGRAMMA CAP.COM

Il programma esemplificativo che segue accetta una lettera in immissione dalla tastiera, la trasforma in maiuscola e la stampa sullo schermo. Per la prima volta, si

farà in modo che il programma in linguaggio assembly accetti un input dall'utente. Cominciamo con la struttura del file .COM:

```
.CODE

        ORG  100H
Start:  JMP  CAP
        ;Area dati
CAP:
        ;Qui va il programma

Exit:   INT    20H

        END  Start
```

E aggiungiamo le istruzioni che accetteranno l'immissione:

```
.CODE

        ORG  100H
Start:  JMP  CAP
        ;Area dati
CAP:    MOV    AH,1     ;Richiesta di immissione da tastiera
        INT    21H      ;dall'INT 21H

Exit:   INT    20H

        END  Start
```

Dopo l'esecuzione dell'istruzione INT 21H, il codice ASCII del carattere battuto si trova in AL. Compito del programma è trasformare in maiuscolo la lettera e stamparla. Esiste un modo molto semplice per trasformare in maiuscolo una lettera: i codici ASCII delle minuscole (per esempio, della a) hanno valori più alti dei codici ASCII per le maiuscole (per esempio, per la A). I codici ASCII da A a Z vanno da 65 a 90; quelli per le minuscole da a a z vanno da 97 a 122. Per trasformare in maiuscolo una lettera basta sottrarre un numero dal suo codice ASCII:

Minuscolo	**Codice**			**Maiuscolo**	**Codice**
a	97	sottrai 32	→	A	65
b	98	sottrai 32	→	B	66
c	99	sottrai 32	→	C	67
.				.	
.				.	
.				.	
z	122	sottrai 32	→	Z	90

Il numero che si deve sottrarre è uguale a ASCII(a) - ASCII(A), cioè 97 - 65 = 32, la distanza tra le due sezioni della tabella dei codici ASCII. Ecco dunque come far

diventare maiuscolo il valore ASCII in AL, mediante una nuova istruzione, SUB (subtract, cioè sottrazione):

```
.CODE

        ORG  100H
Start:  JMP  CAP
        ;Area dati
CAP:    MOV    AH,1     ;Richiesta di immissione da tastiera
        INT    21H      ;dall'INT 21H
        SUB    AL,"a"-"A"   ;Trasforma in maiuscolo il
                                carattere scritto

Exit:   INT    20H

        END  Start
```

LE ISTRUZIONI SUB E ADD

L'istruzione SUB si usa in questo modo:

```
SUB     AL,5
```

Qui si sottrae 5 dal valore contenuto in AL. Analogamente, si può scrivere:

```
SUB     AX,DX
```

In questo caso si sottrae dal valore contenuto in AX il valore contenuto in DX. Il risultato viene memorizzato in AX, mentre DX non viene modificato.
Oltre all'istruzione SUB esiste una ADD, cioè un'istruzione di somma, che si usa in questo modo:

```
ADD     AL,5
ADD     AX,DX
```

ADD e SUB si usano spesso. Inoltre, l'assembler consente di utilizzare anche espressioni come "a" - "A". Per esempio, si può usare una riga di questo tipo:

```
SUB     AL,"a"-"A"
```

e in questo modo quel che si vuol fare diventa molto più chiaro che se si fosse semplicemente scritto:

```
SUB     AL,32
```

Analogamente, sono consentite anche espressioni come "a" + "A".

> ***Consiglio:*** L'assembler capisce espressioni che includono operatori come +, -, / e *. Spesso il loro uso consente di rendere molto più leggibile il codice.

Letto un carattere e trasformatolo in maiuscolo sottraendo A da a, ora bisogna stamparlo. Il servizio 2 dell'interrupt 21H, che stampa un carattere sullo schermo, vuole trovare in DL il codice ASCII del carattere che deve stampare. Ora, in CAP.ASM il codice ASCII si trova ancora nel registro AL (perché il servizio 1 lo ha restituito in quel registro). Bisogna spostare il codice da AL a DL e poi stampare il carattere (listato 10.4).

```
CODE

        ORG  100H
Start:  JMP  CAP
        ;Area dati
CAP:    MOV    AH,1     ;Richiesta di immissione da tastiera
        INT    21H      ;dall'INT 21H
        SUB    AL,"a"-"A"      ;Trasforma in maiuscolo il
                                 carattere scritto
        MOV    DL,AL    ;Preparazione per il servizio 2
        MOV    AH,2     ;Richiesta di emissione carattere
        INT    21H      ;Stampa il carattere
Exit:   INT    20H

        END  Start
```

Listato 10.4 *Il programma CAP.ASM*

In questo modo, CAP.ASM è completo. Si è letto un tasto premuto, tramite il servizio 1 dell'INT 21H; lo si è trasformato in una lettera maiuscola e poi lo si è stampanto con il servizio 2 di INT 21H. Dopo aver scritto il programma, lo si assembli per produrre CAP.COM e poi lo si provi. Quando lo si esegue, si vedrà:

```
D:\>CAP
```

Il programma aspetta che venga premuto un tasto. Non appena si preme il tasto di una lettera, poniamo s, il carattere viene inviato in eco allo schermo e viene stampata una S maiuscola. Poi il programma esce:

```
D:\>CAP
sS
D:\>
```

È sicuramente gratificante ottenere i risultati previsti, ma il programma ha ancora molti problemi. Il più grave probabilmente è: che cosa succede se si preme un tasto che non corrisponde a una lettera minuscola? Verranno stampati caratteri strani, perché è stata prevista solo la possibilità di gestire lettere minuscole.
Il problema può essere eliminato controllando il codice ASCII in ingresso per essere sicuri che rappresenti davvero una lettera minuscola. In altre parole, bisogna controllare che il codice ASCII in ingresso sia compreso fra il valore di a e quello di z.

Questo tipo di controllo ci porta al tema dei salti condizionati, che nel linguaggio assembly sono estremamente importanti, poiché sono quasi l'unica istruzione di salto disponibile.

SALTI CONDIZIONATI

Vogliamo perfezionare CAP.ASM in modo che controlli se il codice ASCII in ingresso è compreso fra a e z; in caso contrario, il programma ha termine. Per effettuare il controllo, si divide il processo in due passi. Innanzitutto, si controlla se il carattere è maggiore o uguale ad a; poi si controlla se è minore o uguale a z. Se ambedue i test vengono superati, si trasforma la lettera in maiuscolo, la si stampa e si esce.

L'ISTRUZIONE CMP

Il confronto fra un valore e un altro valore noto si effettua con l'istruzione di confronto del linguaggio assembly, CMP (dall'inglese "compare"). Per saltare in funzione dei risultati del confronto, si usa poi un *salto condizionato* immediatamente dopo l'istruzione CMP. A differenza di quel che succede in BASIC, in linguaggio assembly i confronti sono processi a due stadi: per esempio, il codice per controllare se il valore che si trova in AL (cioè il codice ASCII letto dalla tastiera) è maggiore o uguale ad "a" è:

```
CODE

        ORG  100H
Start:  JMP  CAP
        ;Area dati
CAP:    MOV    AH,1            ;Richiesta di immissione da
                                tastiera
        INT    21H             ;dall'INT 21H
        CMP    AL,"a"          ;Confronta il codice ASCII in
                                ingresso con "a"
        JB     Exit            ;Se la lettera non è minuscola,
                                esce
        SUB    AL,"a"-"A"      ;Trasforma in maiuscolo il
                                carattere scritto
        MOV    DL,AL           ;Preparazione per il servizio 2
        MOV    AH,2            ;Richiesta di emissione carattere
        INT    21H             ;Stampa il carattere
Exit:   INT    20H

        END  Start
```

Si è confrontato AL con il valore ASCII della a; poi si è fatta seguire immediatamente un'istruzione JB (Jump if Below, cioè "salta se minore"):

```
CMP     AL,"a"   ;Confronta il codice ASCII in ingresso⌉
                  con "a"
JB      Exit     ;Se la lettera non è minuscola, esce
```

Se il confronto dice che il valore in AL è minore del valore della a, si salta all'etichetta Exit alla fine del programma e si esce senza trasformare in maiuscolo. Che cosa succede esattamente? Innanzitutto l'istruzione CMP imposta i flag del microprocessore, poi l'istruzione JB controlla questi flag interni e agisce di conseguenza:

```
CMP     AL,"a"       ;Imposta i flag
JB      Exit         ;Legge i flag
```

> ***Consiglio:*** Oltre a CMP, anche tutte le istruzioni matematiche come ADD, SUB o MUL impostano i flag, perciò dopo queste istruzioni si possono usare i salti condizionati anche senza un'istruzione CMP.

Il passo successivo consiste nel controllare se il codice ASCII è minore o uguale a z. Se il codice è maggiore, anche in questo caso si esce; l'istruzione di salto è JA (Jump if Above, cioè "salta se superiore"). In questo modo si completa il programma CAP.ASM (riportato nel listato 10.5), il nostro primo vero programma in linguaggio assembly: accetta un'immissione, genera un'emissione e addirittura effettua un controllo di correttezza.

```
 CODE

        ORG  100H
Start:  JMP  CAP
        ;Area dati
CAP:    MOV    AH,1            ;Richiesta di immissione da⌉
                                tastiera
        INT    21H             ;dall'INT 21H
        CMP    AL,"a"          ;Confronta il codice ASCII in⌉
                                ingresso con "a"
        JB     Exit            ;Se la lettera non è minuscola,⌉
                                esce
        CMP    AL,"z"          ;Confronta il codice ASCII in⌉
                                ingresso con "z"
        JA     Exit            ;Se la lettera non è minuscola,⌉
                                esce
        SUB    AL,"a"-"A"      ;Trasforma in maiuscolo il⌉
                                carattere scritto
```

continua

Listato 10.5. *Il programma CAP.ASM.*

```
        MOV     DL,AL           ;Preparazione per il servizio 2
        MOV     AH,2            ;Richiesta di emissione carattere
        INT     21H             ;Stampa il carattere
Exit:   INT     20H

        END   Start
```

Listato 10.5. *Il programma CAP.ASM.*

ANCORA SUI SALTI CONDIZIONATI

A questo punto, abbiamo visto le due istruzioni JA e JB, che seguono un'istruzione di confronto CMP e, in funzione del risultato, possono effettuare un salto. In realtà le istruzioni condizionali sono molte di più: esistono anche delle variantidi JA e JB. Esistono JAE (Jump if Above or Equal, salta se superiore o uguale); JBE (Jump if Below or Equal, salta se minore o uguale); JNA (Jump if Not Above, salta se non è maggiore), JNB (Jump if Not Below, salta se non è minore), JNAE (Jump if Not Above or Equal, salta se non è maggiore o uguale) e JNBE (Jump if Not Below or Equal, salta se non è minore o uguale). Tutte queste istruzioni possono essere utilizzate dopo una CMP. Ecco le istruzioni di salto condizionato e i relativi significati;

Salto condizionato	**Significato**
JA	Jump if Above (salta se >)
JB	Jump if Below (salta se <)
JAE	Jump if Above or Equal (salta se >=)
JBE	Jump if Below or Equal (salta se <=)
JNA	Jump if Not Above (salta se non >)
JNB	Jump if Not Below (salta se non <)
JNAE	Jump if Not Above or Equal (salta se non >=)
HNBE	Jump if Not Below or Equal (salta se non <=)
JE	Jump if Equal (salta se =)
JNE	Jump if Not Equal (salta se non =)
JZ	Jump if Zero (salta se il risultato è zero)
JNZ	Jump if Not Zero (salta se il risultato non è zero)
JCXZ	Jump if CX = 0 (salta se CX=0: è usato alla fine dei cicli)

Come si può vedere, c'è un'ampia scelta di istruzioni di salto. Se non fosse così, il linguaggio assembly sarebbe davvero difficile da utilizzare. Così, ci sono salti condizionali per soddisfare quasi tutte le esigenze. Chi non ha mai programmato in assembly dovrà fare un po' di esperienza per riuscire a prendere confidenza con il loro impiego.
Ora mettiamo all'opera quello che abbiamo imparato con l'ultimo esempio del capitolo, un programma che converte numeri esadecimali in numeri decimali.

UN PROGRAMMA PER CONVERTIRE DA ESADECIMALE A DECIMALE

Questo programma sarà un po' più consistente: convertirà numeri esadecimali a quattro cifre in numeri decimali. Si inizia come al solito con la struttura del file .COM:

```
.CODE

        ORG  100H
Start:  JMP  CAP
        ;Area dati
CAP:
        ;Qui va il programma

Exit:   INT    20H

        END  Start
```

Innanzitutto, bisogna leggere il numero esadecimale dalla tastiera. Il DOS fornisce alcuni servizi speciali per l'immissione di stringhe, ed è possibile utilizzarli per configurare in memoria un buffer, con una direttiva DB come questa:

```
BUFFER  DB #, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Per riempire il buffer, si utilizzerà il servizio 0AH (ottiene una stringa) dell'interrupt INT 21H, l'equivalente di LINE INPUT nel linguaggio BASIC. Si imposta il numero (#, sopra) all'inizio del buffer, in modo che sia pari alla lunghezza del buffer. (Il servizio 0AH richiede questo numero per non restituire troppi byte.)

> ***Consiglio:*** Il servizio 0AH imposta sempre l'ultimo byte del buffer al valore ASCII 13 (un ritorno carrello) come marcatore di fine stringa, perciò bisogna lasciare spazio per un carattere in più del numero di caratteri che si attende come input.

Nel secondo byte del buffer il servizio OAH inserirà il numero dei byte effettivamente digitati. Con un po' di attenzione si può configurare il buffer in anticipo etichettando espressamente i suoi byte più importanti:

```
.CODE
        ORG  100H
ENTRY:  JMP  DEHEXER
        BUFFER          DB 5
        NUM_TYPED       DB 0
        ASCII_NUM       DB 3 DUP (0)
        END_NUM         DB 0
        CRLF            DB 0
```

```
DEHEXER:MOV     AH,9
        MOV     DX,OFFSET PROMPT
        INT     21H
        MOV     AH,DAH
        MOV     DX,OFSSET BUFFER
        INT     21H

Exit:   INT     20H
```

Si può osservare l'uso della direttiva DUP:

```
BUFFER          DB 5
NUM_TYPED       DB 0
ASCII_NUM       DB 3 DUP (0)
END_NUM         DB 0
CRLF            DB 0
```

La direttiva fa risparmiare tempo. Questa espressione è uguale a NUM_ASCII DB 0, 0, 0: non è un grosso risparmio per 3 byte, ma lo sarebbe se si dovesse riservare lo spazio per 32.000. Ora il buffer è impostato ed etichettato come indicato in figura 10.14

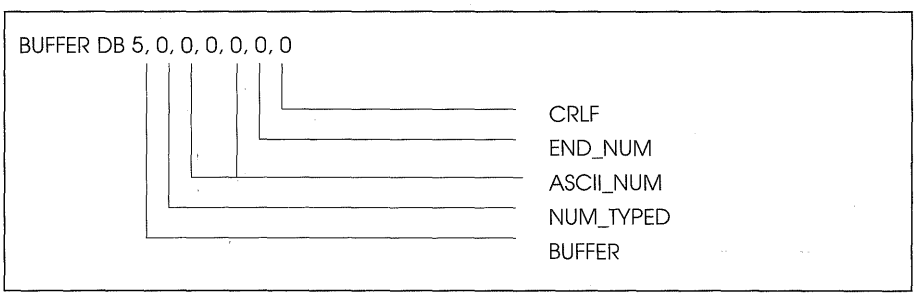

Figura 10.14

Si può anche aggiungere un invito ("Scrivi un numero esadecimale di 4 cifre:$") che venga mostrato sullo schermo attraverso il servizio 9 per la stampa di stringhe. Per stampare questo invito, bisogna passare al servizio 0 un indirizzo di scostamento in DX:

```
.CODE

        ORG  100H
ENTRY:  JMP  DEHEXER
        PROMPT          DB "Scrivi un numero esadecimale
                           di 4 cifre:$"
        BUFFER          DB 5
        NUM_TYPED       DB 0
```

```
        ASCII_NUM           DB 3 DUP (0)
        END_NUM             DB 0
        CRLF                DB 0
DEHEXER:MOV     AH,9
        MOV     DX,OFFSET PROMPT
        INT     21H
        :
```

Sullo schermo il sollecito apparirà in questo modo:

```
Scrivi un numero esadecimale di 4 cifre:
```

Poi, si può usare il servizio 0AH per leggere dalla tastiera il numero esadecimale a 4 cifre, il numero che dovrà essere convertito in decimale e stampato. Si deve semplicemente passare in DX lo scostamento dall'inizio del buffer per il servizio 0AH:

```
.CODE

        ORG  100H
ENTRY:  JMP  DEHEXER
        PROMPT              DB "Scrivi un numero esadecimale
                                di 4 cifre:$"
        BUFFER              DB 5
        NUM_TYPED           DB 0
        ASCII_NUM           DB 3 DUP (0)
        END_NUM             DB 0
        CRLF                DB 0
DEHEXER:MOV     AH,9
        MOV     DX,OFFSET PROMPT
        INT     21H
        MOV     AH,0AH
        MOV     DX,OFFSET BUFFER
        INT     21H
        :
        :
        INT     20H
```

Poi si dà un'istruzione INT 21 H e si accetta il numero esadecimale.

***Nota**: Se si tenta di scrivere un numero di caratteri superiore a quello che può essere accettato nel buffer, il computer emetterà un segnale acustico: è lo stesso "bip" che usa il DOS e, a questo riguardo, si tratta anche dello stesso servizio interno (0AH) che il DOS usa per accettare un'immissione da tastiera al sollecito del sistema.*

Dopo che il buffer si è riempito con l'immissione da tastiera, la stringa ASCII si estende dalle locazioni ASCII_NUM fino a END_NUM. Il <cr> (CHR$(13)) alla fine della stringa restituita andrà nel byte indicato come CRLF e si può ignorare. Il primo passo è quello di convertire la stringa di caratteri in un numero. Se il numero digitato era 1234H, allora il buffer ora appare come in figura 10.15

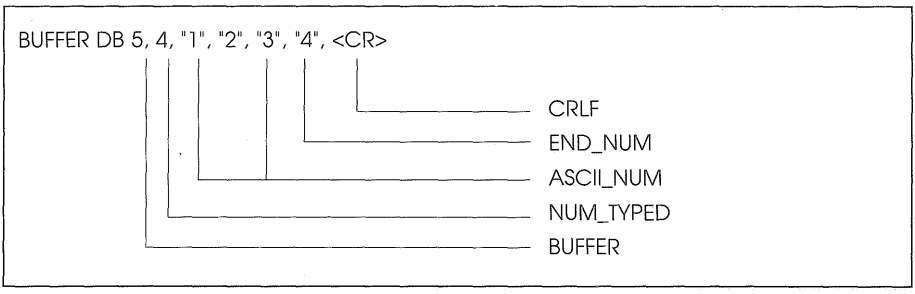

Figura 10.15

Si può semplicemente puntare all'ultimo numero, 4, convertirlo da ASCII in binario, poi puntare al numero successivo, 3, convertirlo in binario, moltiplicarlo per 16 e sommarlo al 4 che già è stato convertito, e via di questo passo procedendo verso le posizioni più significative. In questo modo si cicla su tutti i caratteri.
L'ultima cifra ASCII è stata etichettata come END_NUM, perciò si può leggere il carattere ASCII in quella locazione con l'istruzione MOV AL, END_NUM. Ma come si fa a puntare alle cifre precedenti?
Si può usare come puntatore un registro. Nel linguaggio assembly, il registro BX è stato pensato proprio per essere usato come puntatore: presto si vedrà come. Serve un ciclo per leggere tutte le quattro cifre, perciò si inizia caricando in BX l'indirizzo di scostamento di END_NUM:

```
.CODE
        ORG  100H
ENTRY:  JMP  DEHEXER
        PROMPT              DB "Scrivi un numero esadecimale]
                                di 4 cifre:$"
        BUFFER              DB 5
        NUM_TYPED           DB 0
        ASCII_NUM           DB 3 DUP (0)
        END_NUM             DB 0
        CRLF                DB 0
DEHEXER:MOV    AH,9
        MOV    DX,OFFSET PROMPT
        INT    21H
        MOV    AH,0AH
        MOV    DX,OFFSET BUFFER
        INT    21H
        :
        MOV    BX, OFFSET END_NUM
LOOP1:
        :
        :
```

```
        JB      LOOP1

        INT     20H
        END     ENTRY
```

Ora si carica il carattere ASCII nel registro DL così:

```
.CODE

        ORG  100H
ENTRY:  JMP  DEHEXER
        PROMPT              DB "Scrivi un numero esadecimale
                                di 4 cifre:$"
        BUFFER              DB 5
        NUM_TYPED           DB 0
        ASCII_NUM           DB 3 DUP (0)
        END_NUM             DB 0
        CRLF                DB 0
DEHEXER:MOV     AH,9
        MOV     DX,OFFSET PROMPT
        INT     21H
        MOV     AH,0AH
        MOV     DX,OFFSET BUFFER
        INT     21H
        :
        MOV     BX, OFFSET END_NUM
LOOP1:  MOV     DL, [BX]
        DEC     BX
        :
        :

        JB      LOOP1

        INT     20H
        END     ENTRY
```

Mettendo BX fra parentesi quadre, [BX], si dice al microprocessore che deve utilizzare il valore che trova in BX come un indirizzo. Questa tecnica si chiama *indirizzamento indiretto* e sarà sostanzialmente alla base di tutto quello che faremo nel capitolo 11. [BX] sta per il byte che ora si trova alla locazione END_NUM, perché BX contiene l'indirizzo di scostamento di quel byte (figura 10.16).

E ora si può spostare quel byte in DL con l'istruzione MOV DL, [BX]. Dopo aver spostato il byte in DL, si decrementa di 1 il puntatore BX mediante l'istruzione DEC, che ha lo stesso significato di una SUB BX,1. (L'istruzione corrispondente per incrementare di una unità è INC.) In questo modo DX punta al carattere ASCII precedente: è pronto per il prossimo passaggio nel ciclo (figura 10.17).

Così, dunque, funziona l'indirizzamento indiretto [BX]. BX contiene l'indirizzo di scostamento (nel segmento dei dati) del byte o della parola che si vuole localizzare.

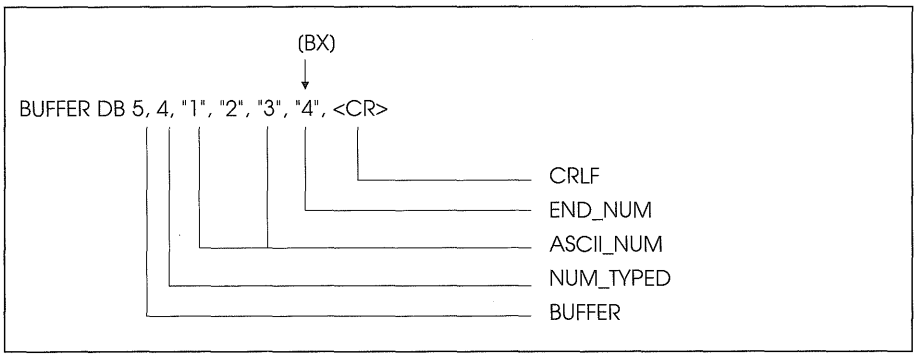

Figura 10.16

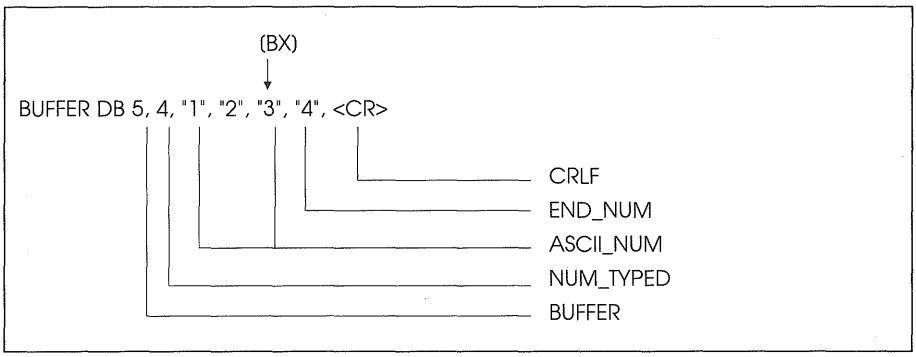

Figura 10.17

Ora che in DL si trova l'ultimo carattere ASCII (4), bisogna convertirlo in binario. Se il numero è nell'intervallo fra 0 e 9, si deve sottrargli il valore ASCII per 0 (in questo modo, 0 diventa 0, 1 diventa 1 e così via). Se è invece nell'intervallo fra A e F, bisogna sottrarre A e sommare 10. Si fa in questo modo:

```
.CODE
        ORG  100H
ENTRY:  JMP  DEHEXER
        PROMPT              DB "Scrivi un numero esadecimale
                                di 4 cifre:$"
        BUFFER              DB 5
        NUM_TYPED           DB 0
        ASCII_NUM           DB 3 DUP (0)
        END_NUM             DB 0
        CRLF                DB 0
```

```
DEHEXER:MOV     AH,9
        MOV     DX,OFFSET PROMPT
        INT     21H
        MOV     AH,0AH
        MOV     DX,OFFSET BUFFER
        INT     21H
        :
        MOV     BX, OFFSET END_NUM
LOOP1:  MOV DX,0
        MOV     DL, [BX]
        DEC     BX
        CMP     DL,"9"
        JBE     UNDER_A
        SUB     DL, "a" - "0" - 10
UNDER_A:SUB     DL, "0"
        :
        JB      LOOP1

        INT     20H
        END     ENTRY
```

DL ora contiene il valore numerico della cifra esadecimale corrente. Per convertire tutto il numero di quattro cifre in decimale, si deve moltiplicare ciascuna cifra per l'opportuna potenza di 16 e sommare il risultato a un totale parziale. Per moltiplicare per 16, si può far *scorrere* il valore in DL verso sinistra.

Esiste un'istruzione in linguaggio assembly, chiamata SHL, che sta per *shift left*, cioè "scorrimento verso sinistra" (esiste anche una SHR per *shift right*, scorrimento verso destra). Ogni volta che si fa scorrere un operando a sinistra di una posizione binaria, è come moltiplicarlo per 2. Si carica in CL il numero di posizioni di cui si vuol far scorrere il valore verso sinistra, poi si fa scorrere verso sinistra il valore che si trova in DL, in questo modo:

```
SHL     DL, CL
```

Per esempio, se in CL c'è un 2 e DL contiene 00000001B, allora, dopo una SHL DL, CL, il registro DL conterrebbe 00000100B. Far scorrere a sinistra di quattro posizioni binarie è come moltiplicare per 16 e, a ogni passaggio nel ciclo, si aggiungerà 4 a CL in modo che ogni volta SHL DL, CL produca il risultato desiderato.

Si badi che quando si fa scorrere DL a sinistra di oltre due posizioni esadecimali il risultato sarà maggiore di quel che può essere contenuto in un byte. Per questo, invece di utilizzare solo DL, si utilizzerà l'intero registro DX:

```
SHL     DX, CL
```

> ***Consiglio:*** Nei microprocessori 80x86 l'istruzione MUL è relativamente lenta, perciò i programmatori spesso cercano di realizzare le loro moltiplicazioni mediante scorrimenti a sinistra e addizioni. Per esempio, per moltiplicare AX per 5, se ne fa una copia in BX, si fa scorrere AX a sinistra di due posizioni, poi gli si somma BX. Questa soluzione è circa dieci volte più veloce di una MUL.

Dopo lo scorrimento, bisogna aggiungere al totale parziale la cifra decimale corrente. Si conserva il totale parziale, per esempio, in AX, e a ogni passaggio nel ciclo si aggiunge DX (listato 10.6).

```
.CODE

        ORG  100H
ENTRY:  JMP  DEHEXER
        PROMPT              DB "Scrivi un numero esadecimale]
                                di 4 cifre:$"
        BUFFER              DB 5
        NUM_TYPED           DB 0
        ASCII_NUM           DB 3 DUP (0)
        END_NUM             DB 0
        CRLF                DB 0DEHEXER:MOV    AH,9
        MOV    DX,OFFSET PROMPT
        INT    21H
        MOV    AH,0AH
        MOV    DX,OFFSET BUFFER
        INT    21H
        :
        MOV    CX,0
        MOV    AX,0
        MOV    BX, OFFSET END_NUM
LOOP1:  MOV DX,0
        MOV    DL, [BX]
        DEC    BX
        CMP    DL,"9"
        JBE    UNDER_A
        SUB    DL, "a" - "0" - 10
UNDER_A:SUB    DL, "0"
        SHL    DX, CL
        ADD    AX, DX
        ADD    CL,4
        CMP    CL,16
        JB     LOOP1
        INT    20H
        END    ENTRY
```

Listato 10.6. *Il programma DEHEXER.ASM, che converte da esadecimale a decimale.*

Ogni volta che si passa nel ciclo, si carica il valore ASCII in DL, lo si trasforma in un numero binario, lo si fa scorrere verso sinistra e lo si somma al totale parziale in AX. Il ciclo viene eseguito solo quattro volte, una per ciascuna cifra.
A questo punto, la stringa ASCII è stata convertita in binario, e il suo valore si trova in AX. Bisogna ancora convertirla in cifre ASCII decimali. Lo si fa estraendo progressivamente le cifre decimali, attraverso divisioni successive per 10 del numero che si trova in AX. Ogni volta che si divide per 10, il resto è una cifra decimale. Per esempio, se si divide 21 per 10, il risultato è 2 con resto 1.

LE ISTRUZIONI DIV E MUL

Nel linguaggio assembly esiste un'istruzione per la divisione, DIV. Se si carica in AX il numero da dividere e si divide per un registro lungo un byte, in questo modo:

```
DIV BL
```

l'istruzione divide AX per BL. Si assume che AX contenga il numero da dividere, quando si divide per un byte. Il quoziente viene restituito in AL e il resto va in AH. Invece, se si utilizza questa istruzione (con un registro da 16 bit):

```
DIV BX
```

il microprocessore assume che si voglia dividere il numero (word doppia) in DX:AX per il registro specificato, BX.

***Nota**: La notazione DX:AX è una notazione poco brillante per specificare le word doppie, visto che anche gli indirizzi sono specificati con un "due punti"; tuttavia, quando si usano registri di segmento, si può stare tranquilli che non si tratta di un indirizzo.*

Analogamente, MUL BL moltiplica AL per BL e mette il risultato in AX; MUL BX moltiplica BX per AX e mette il risultato in DX:AX.

> ***Consiglio:*** È una buona idea pianificare in anticipo l'uso dei registri, in modo che le istruzioni come MUL lascino i risultati nei registri desiderati e non si debbano spostare molto spesso i dati da un registro all'altro, come invece capita spesso nei programmi in linguaggio assembly.

Se si usa l'istruzione DIV BX, il numero in DX:AX verrà diviso per BX, perciò si carica in DX 0 e in BX 10. AX contiene già il numero da convertire. Finita la divisione, AX conterrà il quoziente (pronto per essere diviso di nuovo per 10 nel passo successivo, al fine di estrarre la successiva cifra decimale) e DX contiene il resto:

```
CODE

        ORG  100H
ENTRY:  JMP  DEHEXER
        PROMPT              DB "Scrivi un numero esadecimale⌉
                                di 4 cifre:$"
        BUFFER              DB 5
        NUM_TYPED           DB 0
        ASCII_NUM           DB 3 DUP (0)
        END_NUM             DB 0
        CRLF                DB 0
DEHEXER:MOV    AH,9
        MOV    DX,OFFSET PROMPT
        INT    21H
        MOV    AH,0AH
        MOV    DX,OFFSET BUFFER
        INT    21H
        :
        MOV    CX,0
        MOV    AX,0
        MOV    BX, OFFSET END_NUM
LOOP1:  MOV DX,0
        MOV    DL, [BX]
        DEC    BX
        CMP    DL,"9"
        JBE    UNDER_A
        SUB    DL, "a" - "0" - 10
UNDER_A:SUB    DL, "0"
        SHL    DX, CL
        ADD    AX, DX
        ADD    CL,4
        CMP    CL,16
        JB     LOOP1

        MOV    CX,0
        MOV    BX, 10
LOOP2:  MOV    DX,0
        DIV    BX          ; Dopo questa istruzione, DX⌉
                             contiene la cifra decimale corrente
        INT    20H
        END    ENTRY
```

Quello che si vuole è il resto in DX, la cifra decimale corrente. Si badi che le cifre vengono estratte in ordine inverso. Per esempio, se il numero decimale in AX fosse 4321, dividendo per 10 la prima volta si otterrebbe un resto 1, la seconda volta un resto 2 e così via.

Per memorizzare queste cifre decimali che ora sono in DX, si utilizza lo stack, mediante l'istruzione PUSH DX. Lo stack è una parte di memoria riservata per

contenere dati e, nel prossimo capitolo, il BASIC inserirà valori sullo stack per la lettura. Vediamo come funziona lo stack. Se ci fosse un valore 5 in DX e si eseguisse l'istruzione PUSH DX, lo stack sarebbe come in figura 10.18 (si possono inserire nello stack con PUSH word e non byte).

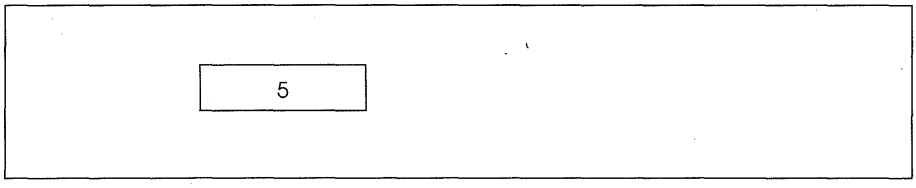

Figura 10.18

Poi si può inserire nello stack CX, che supponiamo contenga un valore 3 (figura 10.19).

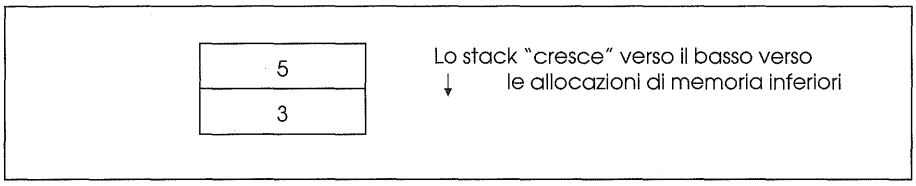

Figura 10.19

Dopo di questo si può inserire AX, che contiene uno 0 (figura 10.20).

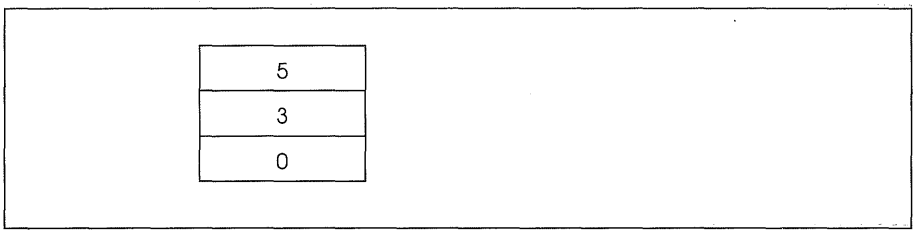

Figura 10.20

Poi si potrebbe dare l'istruzione POP BX per estrarre valori dallo stack. L'ultimo valore che è stato inserito sullo stack (0) verrà estratto e messo in BX, e lo stack sarà come in figura 10.21.

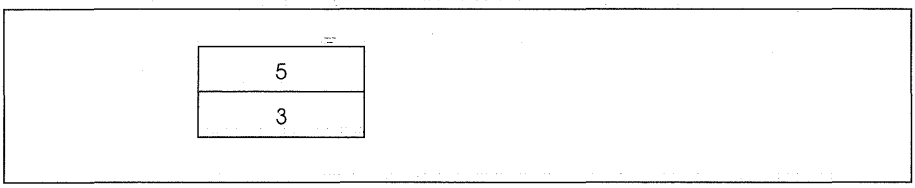

Figura 10.21

Con un'altra istruzione POP, per esempio POP DX, si estrae 3 e lo si depone in DX, e lo stack risulta come in figura 10.22.

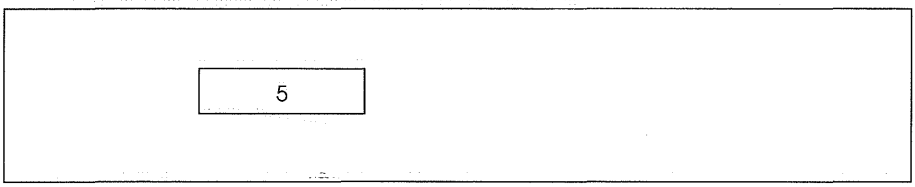

Figura 10.22

POP CX caricherebbe quest'ultimo valore in CX e lo stack resterebbe vuoto. Si osservi che i valori vengono estratti dallo stack nell'ordine inverso a quello in cui vi sono stati depositati (una caratteristica che ci avvantaggerà). Nel nostro DEHEXER, le singole cifre decimali verranno estratte dal valore in AX e depositate con una PUSH sullo stack. un valore decimale 8573 verrebbe depositato nello stack nell'ordine 3, 7, 5, 8 (si terrà traccia anche di quanti numeri vengono inseriti nello stack, in modo da poterli poi estrarre con l'istruzione POP: un numero esadecimale di quattro cifre può dare un numero decimale con un numero di cifre variabile da 1 a 5).

***Nota**: Quando vengono estratte, le cifre compaiono nell'ordine in cui le si vuole stampare: 8, 5, 7 e 3.*

Ecco come si caricano sullo stack le singole cifre decimali:

```
CODE

        ORG   100H
ENTRY:  JMP   DEHEXER
        PROMPT              DB "Scrivi un numero esadecimale]
                                di 4 cifre:$"
        BUFFER              DB 5
        NUM_TYPED           DB 0
        ASCII_NUM           DB 3 DUP (0)
```

```
        END_NUM                 DB 0
        CRLF                    DB 0
DEHEXER:MOV     AH,9
        MOV     DX,OFFSET PROMPT
        INT     21H
        MOV     AH,0AH
        MOV     DX,OFFSET BUFFER
        INT     21H
        :
        MOV     CX,0
        MOV     AX,0
        MOV     BX, OFFSET END_NUM
LOOP1:  MOV DX,0
        MOV     DL, [BX]
        DEC     BX
        CMP     DL,"9"
        JBE     UNDER_A
        SUB     DL, "a" - "0" - 10
UNDER_A:SUB     DL, "0"
        SHL     DX, CL
        ADD     AX, DX
        ADD     CL,4
        CMP     CL,16
        JB      LOOP1

        MOV     CX,0
        MOV     BX, 10
LOOP2:  MOV     DX,0
        DIV     BX
        PUSH    DX
        INC     CX              ;CX contiene il numero delle
                                 cifre caricate sullo stack
        CMP     AX,0            ;Rimane ancora un numero da
                                 cui togliere cifre?
        JA      LOOP2           ;Se sì, si rientra nel ciclo
        :
        INT     20H
        END     ENTRY
```

A questo punto abbiamo quasi finito. Le cifre decimali sono sullo stack e il loro numero sta in CX. Si può far stampare un messaggio che dica "Questo numero in decimale è:", utilizzando il servizio 9 dell'INT 21H:

```
CODE

        ORG  100H
ENTRY:  JMP  DEHEXER
        PROMPT                  DB "Scrivi un numero esadecimale
                                    di 4 cifre:$"
```

```
        BUFFER              DB 5
        NUM_TYPED           DB 0
        ASCII_NUM           DB 3 DUP (0)
        END_NUM             DB 0
        CRLF                DB 0
        ANS_STRING          DB 13, 10, "Questo numero in
                                       decimale è: $"
DEHEXER:MOV     AH,9
        MOV     DX,OFFSET PROMPT
        INT     21H
        MOV     AH,0AH
        MOV     DX,OFFSET BUFFER
        INT     21H
        :
        MOV     CX,0
        MOV     AX,0
        MOV     BX, OFFSET END_NUM
LOOP1:  MOV DX,0
        MOV     DL, [BX]
        DEC     BX
        CMP     DL,"9"
        JBE     UNDER_A
        SUB     DL, "a" - "0" - 10
UNDER_A:SUB     DL, "0"
        SHL     DX, CL
        ADD     AX, DX
        ADD     CL,4
        CMP     CL,16
        JB      LOOP1

        MOV     CX,0
        MOV     BX, 10
LOOP2:  MOV     DX,0
        DIV     BX
        PUSH    DX
        INC     CX
        CMP     AX,0
        JA      LOOP2

        MOV     AH,9
        MOV     DX,OFFSET ANS_STRING
        INT     21H
                :
        INT     20H
        END     ENTRY
```

E poi si possono semplicemente stampare le cifre con il servizio 2 di INT 21H e istruzioni POP DX. Si osservi che, avendo estratto le cifre dal numero di partenza in

ordine inverso, poiché lo stack ha invertito nuovamente l'ordine, le cifre possono essere stampate nell'ordine in cui vengono estratte dallo stack. In altre parole, se il numero fosse stato 1234, si sarebbe ottenute le cifre, mediante le divisioni successive, nell'ordine 4, 3, 2 e 1, cosicché lo stack sarebbe come in figura 10.23

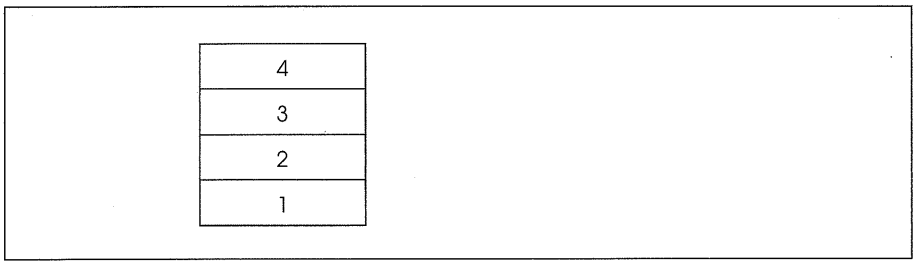

Figura 10.23

La prima cifra che si estrarrebbe dallo stack, e che si stamperebbe, sarebbe 1, seguita da 2 e via di questo passo: si stamperebbe quindi 1234. Le singole cifre devono ancora essere convertite in ASCII. Per esempio, se si estrae dallo stack 1, questo ancora non è l"1" ASCII. Per convertirlo in ASCII, bisogna aggiungerli il valore ASCII dello "0", in modo che 0 diventi "0", 1 diventi "1" e via dicendo.
Ecco il ciclo in cui si estraggono le cifre dallo stack. Le cifre vengono depositate nel registro DX, poiché il servizio 2 vuole che i caratteri ASCII da stampare si trovino in DL, e si usa la nuova istruzione LOOP:

```
CODE

        ORG  100H
ENTRY:  JMP  DEHEXER
        PROMPT              DB "Scrivi un numero esadecimale
                                di 4 cifre:$"
        BUFFER              DB 5
        NUM_TYPED           DB 0
        ASCII_NUM           DB 3 DUP (0)
        END_NUM             DB 0
        CRLF                DB 0
        ANS_STRING          DB 13, 10, "Questo numero in
                                       decimale è: $"
DEHEXER:MOV    AH,9
        MOV    DX,OFFSET PROMPT
        INT    21H
        MOV    AH,0AH
        MOV    DX,OFFSET BUFFER
        INT    21H
        :
```

```
        MOV     CX,0
        MOV     AX,0
        MOV     BX, OFFSET END_NUM
LOOP1:  MOV     DX,0
        MOV     DL, [BX]
        DEC     BX
        CMP     DL,"9"
        JBE     UNDER_A
        SUB     DL, "a" - "0" - 10
UNDER_A:SUB     DL, "0"
        SHL     DX, CL
        ADD     AX, DX
        ADD     CL,4
        CMP     CL,16
        JB      LOOP1
        MOV     CX,0
        MOV     BX, 10
LOOP2:  MOV     DX,0
        DIV     BX
        PUSH    DX
        INC     CX
        CMP     AX,0
        JA      LOOP2
        MOV     AH,9
        MOV     DX,OFFSET ANS_STRING
        INT     21H
        MOV     AH,2
LOOP3:  POP     DX
        ADD     DX,"0"
        INT     21H
        LOOP    LOOP3

        INT     20H
        END     ENTRY
```

L'ISTRUZIONE LOOP

L'istruzione LOOP è come un'istruzione di ciclo FOR...NEXT in BASIC. Ecco come si usa LOOP per ciclare sulle cifre inserite nello stack, estrarle e stamparle. Per usare LOOP, basta inserire in CX il numero delle volte che deve essere eseguito un ciclo, definire un'etichetta e procedere in questo modo:

```
        MOV     CX,5
LOOP_1
        :
        :
        LOOP    LOOP_1
```

Il questo caso, il corpo di LOOP_1 verrà eseguito cinque volte.
In DEHEXER, il ciclo precedente (in cui si estraevano le cifre decimali dal valore in AX) ha lasciato già in CX il numero delle cifre, perciò l'indice dell'iterazione è già impostato. Tutto quel che si deve fare è usare LOOP, e stampare le cifre man mano che vengono estratte dallo stack:

```
        MOV     AH,2
LOOP3:  POP     DX
        ADD     DX,"0"
        INT     21H
        LOOP    LOOP3
```

Ed è tutto. Si è letto un numero esadecimale, lo si è convertito in binario moltiplicando ciascuna cifra per una potenza di 16, si sono estratte le cifre decimali mediante una serie di divisioni successive per 10, se ne è invertito l'ordine e poi le cifre sono state stampate. Sono stati fatti molti progressi.
Il programma funziona e lo si può mettere alla prova. Accetta numeri esadecimali di quattro cifre e stampa la versione decimale corretta. C'è però ancora una differenza tra questo programma e la maggior parte dei file .ASM: questi in genere hanno almeno una procedura definita al loro interno. Sarà questo l'argomento finale del capitolo, un argomento peraltro indispensabile per quello che si farà nel capitolo 11.

PROCEDURE IN LINGUAGGIO ASSEMBLY

In BASIC, si possono scrivere subroutine, sottoprogrammi e funzioni. In linguaggio assembly si possono scrivere solo procedure. Si può trasformare DEHEXER in una singola procedura aggiungendo le direttive PROC e ENDP, che delimiteranno il codice in questo modo:

```
.CODE

        ORG   100H
ENTRY:  JMP   DEHEXER
        PROMPT                DB "Scrivi un numero esadecimale
                                 di 4 cifre:$"
        :
DEHEXER PROC
        MOV     AH,9
        MOV     DX,OFFSET PROMPT        La procedura DEHEXER
        :
        INT     20H
DEHEXER ENDP
        END     ENTRY
```

Ecco come appare ora il programma nel suo complesso:

```
.CODE

        ORG  100H
ENTRY:  JMP  DEHEXER
        PROMPT              DB "Scrivi un numero esadecimale]
                                di 4 cifre:$"
        BUFFER              DB 5
        NUM_TYPED           DB 0
        ASCII_NUM           DB 3 DUP (0)
        END_NUM             DB 0
        CRLF                DB 0
        ANS_STRING          DB 13, 10, "Questo numero in]
                                       decimale è: $"
DEHEXER PROC
        MOV    AH,9
        MOV    DX,OFFSET PROMPT
        INT    21H
        MOV    AH,0AH
        MOV    DX,OFFSET BUFFER
        INT    21H
        :
        MOV    CX,0
        MOV    AX,0
        MOV    BX, OFFSET END_NUM
LOOP1:  MOV    DX,0
        MOV    DL, [BX]
        DEC    BX
        CMP    DL,"9"
        JBE    UNDER_A
        SUB    DL, "a" - "0" - 10
UNDER_A:SUB    DL, "0"
        SHL    DX, CL
        ADD    AX, DX
        ADD    CL,4
        CMP    CL,16
        JB     LOOP1

        MOV    CX,0
        MOV    BX, 10
LOOP2:  MOV    DX,0
        DIV    BX
        PUSH   DX
        INC    CX
        CMP    AX,0
        JA     LOOP2
        MOV    AH,9
```

```
        MOV     DX,OFFSET ANS_STRING
        INT     21H

        MOV     AH,2
LOPP3:  POP     DX
        ADD     DX,"0"
        INT     21H
        LOOP    LOOP3

        INT     20H
DEHEXER ENDP

        END     ENTRY
```

Si sono aggiunte le direttive PROC e ENDP, che definiscono le procedure. La direttiva PROC comunica all'assembler che si vuole definire una procedura, e la direttiva ENDP indica che la definizione della procedura è finita.
Se non si è nella procedura principale si deve concludere la procedura con una istruzione RET (return, ritorna). Si può suddividere DEHEXER in due procedure per vedere come si fa. Si può, per esempio, stampare il numero decimale che costituisce la soluzione in una nuova procedura che si può chiamare PRINT_NUM.
Non appena si è decodifica il numero esadecimale in ingresso e lo si è depositato in AX come valore binario, si può chiamare PRINT_NUM perché estragga da AX le cifre decimali, le collochi sullo stack e poi le stampi. Si chiama PRINT_NUM con l'istruzione CALL PRINT_NUM, e alla fine si ritorna con un'istruzione RET (proprio come si mette RETURN alla fine di una subroutine BASIC). Il procedimento è illustrato in figura 10.24.

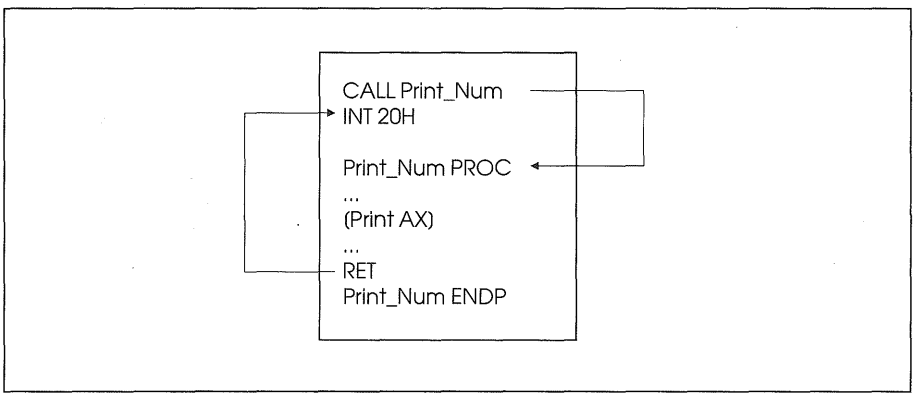

Figura 10.24

La struttura del programma diventa questa:

```
 .CODE
        ORG    100H
ENTRY:  JMP DEHEXER
        ;Dati
DEHEXER  PROC
          :
          :                          Procedura
         CALL    PRINT_NUM           DEHEXER
         INT     20H
DEHEXER  ENDP

PRINT_NUM    PROC
          :                          Procedura
          :                          PRINT_NUM
          RET
PRINT_NUM    ENDP

          END    ENTRY
```

Funziona come previsto. Quando si chiama PRINT_NUM con la CALL, il controllo viene trasferito alla prima riga di questa procedura. L'esecuzione procede finché non viene raggiunta l'istruzione di ritorno, RET; a quel punto il controllo viene restituito alla riga immediatamente successiva all'istruzione CALL PRINT_NUM nella procedura principale. Non si chiamano mai procedure con argomenti (per esempio, CALL PRINT_NUM(AX)); i valori vengono passati invece nei registri. Il programma completo è risportato nel listato 10.7.

```
.CODE

        ORG 100H
ENTRY:  JMP DEHEXER
        PROMPT          DB "Scrivi un numero esadecimale
                            di 4 cifre:$"
        BUFFER          DB 5
        NUM_TYPED       DB 0
        ASCII_NUM       DB 3 DUP (0)
        END_NUM         DB 0
        CRLF            DB 0
        ANS_STRING      DB 13, 10, "Questo numero in decimale
                                   è: $"
DEHEXER PROC
        MOV     AH,9
        MOV     DX,OFFSET PROMPT
        INT     21H
        MOV     AH,0AH
```

continua

Listato 10.7. *DEHEXER.ASM con una subroutine*

```
        MOV     DX,OFFSET BUFFER
        INT     21H

        MOV     CX, 0
        MOV     AX,0
        MOV     BX, OFFSET END_NUM
LOOP1:  MOV     DX,0
        MOV     DL, [BX]
        DEC     BX
        CMP     DL,"9"
        JBE     UNDER_A
        SUB     DL, "a" - "0" - 10
UNDER_A:SUB     DL, "0"
        SHL     DX, CL
        ADD     AX, DX
        ADD     CL,4
        CMP     CL,16
        JB      LOOP1

        CALL    PRINT_NUM

        INT     20H
DEHEXER ENDP

PRINT_NUM       PROC
        MOV     CX,0
        MOV     BX, 10
LOOP2:  MOV     DX,0
        DIV     BX
        PUSH    DX
        INC     CX
        CMP     AX,0
        JA      LOOP2

        MOV     AH,9
        MOV     DX,OFFSET ANS_STRING
        INT     21H

        MOV     AH,2
LOOP3:  POP     DX
        ADD     DX,"0"
        INT     21H
        LOOP    LOOP3
        RET
PRINT_NUM       ENDP
        END     ENTRY
```

Listato 10.7. *DEHEXER.ASM con una subroutine.*

Le procedure in linguaggio assembly non restituiscono specificamente dei valori, come possono fare invece le funzioni in BASIC. Le informazioni vengono restituite nei registri oppure, in qualche caso, nei flag.

Consiglio: Non esiste, in linguaggio assembly, l'equivalente di una variabile locale formale. Tutte le variabili che si trovano nello stesso modulo sono condivise. Si possono però rendere locali le variabili inserendole in un altro file.

CONCLUSIONE

Questa è la nostra veloce introduzione al linguaggio assembly. Raggiunto questo punto, tutte le conoscenze acquisite verranno messe all'opera nell'interfacciamento del BASIC al linguaggio assembly, che costituisce l'argomento del capitolo 11.

CAPITOLO 11

INTERFACCIAMENTO TRA BASIC E LINGUAGGIO ASSEMBLY

In questo capitolo si imparerà:

- come collegare al BASIC procedure in linguaggio assembly;
- come lavorare con dati BASIC in linguaggio assembly;
- come leggere parametri passati dal BASIC;
- come restituire valori al BASIC.

Collegando al BASIC del codice in linguaggio assembly è possibile rendere più efficienti i programmi e aumentarne la velocità di elaborazione. Verranno presentati numerosi esempi in linguaggio assembly sia per funzioni che per sottoprogrammi BASIC, che in BASIC appariranno come normali sottoprogrammi e normali funzioni, benché siano scritti in linguaggio assembly. Per esempio, il listato 11.1 mostra come risulterebbe, in linguaggio assembly, una funzione BASIC che somma due interi:

```
.MODEL MEDIUM, BASIC
.CODE
.286
        PUBLIC Addem
Addem   PROC FAR USES DI SI DS ES, VALUE1:WORD, VALUE2:WORD
        MOV BX, VALUE1  ;Prende l'indirizzo di VALUE dallo
                        ;stack (il BASIC passa il riferimento)
        MOV AX, [BX]
        MOV BX, VALUE2
```

continua

Listato 11.1. *Versione in linguaggio assembly di funzione BASIC che somma due interi*

```
        ADD AX, [BX]
        RET
Addem   ENDP
        END
```

Listato 11.1. *Versione in linguaggio assembly di funzione BASIC che somma due interi*

Volendo, si possono usare gli esempi di questo capitolo come modelli, mantenendo immutata l'interfaccia del codice in linguaggio assembly verso il BASIC, adattando invece il corpo dei programmi alle proprie esigenze.
Va comunque ricordato che questo è un argomento avanzato: buona parte di quel che segue non è immediato.

COLLEGAMENTO AL BASIC

Per poter utilizzare questo capitolo, è necessario avere a disposizione un assembler Microsoft, per esempio il MASM 5.1, per assemblare il codice che verrà sviluppato. Si può avere, per esempio, un file A.ASM che contiene una procedura in linguaggio assembly denominata PrintString( ). La prima cosa da fare è assemblare il file:

```
D:\>MASM A;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.

  49894 + 311895 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

Questo produce A.OBJ. Se si usa il QuickBasic, esiste solo un metodo per rendere disponibile ai propri programmi PrintString( ): bisogna effettuare il link di A.OBJ in una Quick Library e caricare quella libreria quando si avvia il QuickBasic.
Per produrre una Quick Library con il nome A.QLB, questo è il metodo:

```
D:\> LINK /Q A.OBJ QB.LIB, A.QLB,, BQL845.LIB;
```

Una volta creato A.QLB, si può semplicemente avviare QB in questo modo (dove PROGRAM.BAS è il programma da cui si vuole chiamare PrintString( )):

```
D:\> QB PROGRAM /L A
```

Il comando /L carica in memoria la Quick Library A.QLB, pronta per essere utilizzata.
In QuickBASIC esteso, si costruisce la Quick Library in questo modo:

```
D:\> LINK /Q A.OBJ QBX.LIB, A.QLB,, QBXQLB.LIB;
```

Una volta creata A.QLB, si avvia ABX in questo modo:

```
D:\> QBX PROGRAM /L A
```

Si può anche usare il compilatore dalla riga dei comandi, BC.EXE (qualsiasi versione, la 4.5 o la 7.10, per esempio). In questo caso, si comincia assemblando A.ASM, il codice sorgente in linguaggio assembly per PrintString( ):

```
D:\> MASM A;
Microsoft (R) Macro Assembler Version 5.10
Copyright (C) Microsoft Corp 1981, 1988. All rights reserved.

  49894 + 311895 Bytes symbol space free

       0 Warning Errors
       0 Severe  Errors
```

Questo crea A.OBJ; poi, si compila il programma BASIC che chiama PrintString( ):

```
D:\> BC PROGRAM;

Compiler Microsoft (R) QuickBASIC Versione 4.50
(C] Copyright Microsoft Corporation 1982-1989.
Tutti i diritti riservati.

43548 Byte disponibili
43722 Byte liberi

     0 Errori di avvertenza
     0 Errori gravi
```

In questo modo si crea PROGRAM.OBJ. Poi si collegano i due file .OBJ con LINK (si tenga presente che bisogna collegare anche tutte le altre librerie eventualmente richieste dal codice di PROGRAM):

```
D:\> LINK PROGRAM + A;
Microsoft (R) Overlay Linker versione 3.69
Copyright (C) Microsoft Corp 1983-1988.  Tutti i diritti
riservati.
```

Così si crea un file denominato PROGRAM.EXE, pronto per l'esecuzione.

***Nota**: Con il BASIC PDS 7.10 viene fornito uno strumento denominato Programmer's Work Bench (PWB), progettato per semplificare il lavoro in ambienti di linguaggi differenti. Sfortunatamente, fin qui è stato fornito solo il supporto per collegare il BASIC con il C, ma non con il linguaggio assembly.*

Prima di iniziare, si deve anche notare che il compilatore QBX.EXE talvolta gestisce i dati (stringhe e matrici) in modo un po' diverso sia da BC.EXE sia da QB.EXE. Per

questo alcuni degli esempi che seguono saranno suddivisi in due parti, una per BC.EXE e QB.EXE e l'altra per QBX.EXE.

PASSAGGIO DEI PARAMETRI DAL BASIC

Come tutti i linguaggi, il BASIC passa i parametri per sottoprogrammi e funzioni sullo stack. Ogni parametro viene inserito sullo stack nell'ordine in cui appare nella chiamata; per esempio, se si usa MID$ in questo modo:

```
MyChar$ = MID$(MyString$, Place%, Num%)
```

prima di passare il controllo a MID$, MyString$ viene inserita sullo stack, seguita da Place% e poi da Num%. Il passaggio dei parametri in quest'ordine viene denominato *convenzione di chiamata BASIC.*

Ogni parametro è passato, in effetti, per *riferimento vicino* (a meno che si usino le parole chiave BYVAL o SEG). Questo significa che il BASIC in effetti passa l'indirizzo di offset (una word) della variabile in memoria. Per esempio, se si disponesse di una funzione chiamata Add5%( ), che somma 5 a un intero passatole come argomento, e la si chiamasse in questo modo:

```
Val% = 9
PRINT "9 + 5 ="; Add5%(Val%)
```

allora il BASIC passerebbe sullo stack l'*indirizzo di offset* di Val% (rispetto all'inizio del segmento dei dati) prima di chiamare Add5%( ). Quando si arriva a Add5%( ), lo stack è organizzato come in figura 11.1

Qui Indirizzo di ritorno è l'indirizzo (due word) a cui viene restituito il controllo quando Add5%( ) ha completato la sua attività. Resta da vedere come si possa leggere l'argomento passato, Val%.

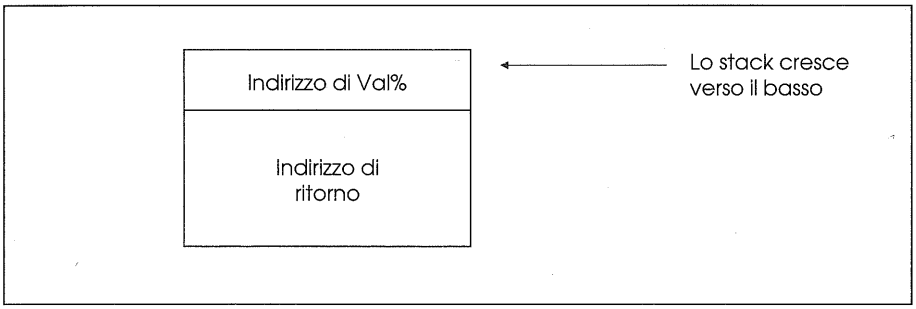

Figura 11.1

Un tempo, chi programmava in linguaggio assembly doveva leggere i valori come l'indirizzo di Val% direttamente dallo stack, ma la nuova direttiva estesa PROC (sotto MASM 5.1) evita questo problema. Si può invece iniziare il codice in linguaggio assembly per Add5 nel modo seguente, con la nuova definizione estesa di PROC:

```
Add5    PROC FAR USES DI SI DS ES, VALUE1:WORD
        :
        :
```

Qui si dice all'assembler che nel programma si useranno i registri DI, SI, DS e ES (non è vero, non verranno realmente usati: è solo un caso dimostrativo). L'assembler quindi aggiunge automaticamente all'inizio del codice per caricare dei valori in questi registri, e alla fine per estrarli, in modo che i valori originali utilizzati dal BASIC vengano conservati. Si indica inoltre che al programma verrà passato il parametro VALUE1, una word.
Da questo punto in poi, si può fare riferimento a VALUE1 come si farebbe riferimento a qualsiasi altra variabile. Cioè, per ottenere l'indirizzo dell'intero passato in BX, è sufficiente fare così:

```
Add5    PROC FAR USES DI SI DS ES, VALUE1:WORD
        MOV BX, VALUE1    ;Prende dallo stack l'indirizzo di
                           VALUE1 (il BASIC passa il riferimento)
        :
```

Si ricorda che ciò che il BASIC trasferisce è l'*indirizzo* del parametro e non il parametro attivo. Poiché ogni indirizzo corrisponde a un indirizzo di offset, vale a dire, una word, tutti i parametri trasferiti avranno la lunghezza di una word. Questi indirizzi di offset verranno utilizzati per ottenere il parametro corrente in AX al quale, successivamente, verrà aggiunto 5:

```
Add5    PROC FAR USES DI SI DS ES, VALUE1:WORD
        MOV BX, VALUE1    ;Prende dallo stack l'indirizzo diß
                           VALUE1 (il BASIC passa il riferimento)
        MOV AX, [BX]
        ADD AX, 5
        :
```

Il passo successivo è restituire il risultato al BASIC: questo procedimento sarà approfondito nel prossimo paragrafo. In breve, una funzione restituisce al BASIC valori di tipo INTEGER caricandoli nel registro AX. Il risultato è già in AX, perciò è sufficiente aggiungere una istruzione di ritorno:

```
Add5    PROC FAR USES DI SI DS ES, VALUE1:WORD

        MOV BX, VALUE1    ;Prende dallo stack l'indirizzo di
                           VALUE1 (il BASIC passa il riferimento)
        MOV AX, [BX]
        ADD AX, 5
```

```
        RET
Add5    ENDP
        END
```

Questo è tutto. Si è caricato in AX il valore passato, si è sommato 5 e si è lasciato in AX il risultato, che costituisce il valore restituito dalla funzione. Il programma chiamante leggerà questo valore dal registro. In questo capitolo si farà uso spesso della definizione di PROC estesa per estrarre dallo stack i parametri passati. In effetti, la definizione di PROC estesa renderà le procedure molto semplici.

RESTITUZIONE DI VALORI AL BASIC

A questo punto ci si chiederà come fanno le funzioni BASIC a restituire valori al programma chiamante? Il procedimento è abbastanza semplice. Ecco come vengono restituiti i valori, in funzione del tipo di dato:

Tipo di dato	*Restituito*
INTEGER	In AX
LONG	In DX:AX (word alta in DX, bassa in AX)
Tutti gli altri	Indirizzo di offset in AX

In altre parole, per restituire un valore INTEGER, si colloca l'intero nel registro AX e si esce. Per restituire un intero LONG, si carica la word più significativa in DX, la meno significativa in AX e si esce.
Per tutti gli altri tipi di dati, si deve restituire in AX l'indirizzo di offset del dato, il che può essere un po' contorto. Se si pensa di restituire un dato di tipo diverso da INTEGER o LONG, si devono seguire determinati passi. Il BASIC, quando chiama la funzione, passa sullo stack l'indirizzo a cui vanno memorizzati i dati restituiti. Quando si recupera il controllo, questo indirizzo si trova sullo stack a [BP+6], dove BP è un registro speciale, denominato *puntatore alla base* (*base pointer*).
Nel capitolo precedente si è visto come funzioni il modo di indirizzamento indiretto [BX]. [BP] funziona nello stesso modo, salvo che BP punta a una locazione sullo stack. In effetti non si vuole il valore in [BP], ma il valore all'indirizzo posto sei byte più in là, cioè a BP + 6. L'assembler permette di fare riferimento a tale valore con la notazione [BP+6] (sono consentiti anche valori negativi, come [BP-6] o [BX-2]). Ecco, dunque, i passi da seguire per restituire valori di tipo diverso da INTEGER o LONG:

1. Quando si ottiene il controllo, si memorizza il valore in [BP+6] (l'offset del valore di ritorno), perché sia possibile utilizzarlo in seguito. Per esempio, si può caricare in DX quell'offset in questo modo: MOV DX, [BP+6].

2. Quando si è pronti per restituire il dato, si colloca il valore da restituire a quell'offset in memoria.
3. Infine, si colloca quell'offset in AX e si ritorna.

Questo argomento verrà trattato nei dettagli, nel paragrafo "Funzioni che restituiscono una stringa". Nel frattempo, si vedrà come comportarsi con i dati ricevuti dal BASIC.

LA RAPPRESENTAZIONE INTERNA DEI DATI DEL BASIC

Per chi programma in linguaggio assembly la vera difficoltà non sta tanto nel ricevere o restituire dati, quanto nel decodificare i valori passati. Per esempio, se viene passato un valore SINGLE e dalla memoria viene letto il valore &H23110000, come si fa a sapere quale numero rappresentino questi quattro byte? Per poter leggere e passare dati dal BASIC e al BASIC, bisogna conoscere esattamente la rappresentazione interna dei dati nel BASIC, cioè come fa il BASIC a memorizzare i suoi dati.
Questi sono i tipi di dati più usati nel BASIC:

Tipo	**Simbolo**	**Byte**	**Campo di variabilità**
INTEGER	%	2	da -32768 a 32767
LONG	&	4	da -2147483648 a 2147483647
SINGLE	!	4	da -3.402823E+38 a -1.40129E-45
DOUBLE	#	8	da -1.79769313486232D308 a -2.2250738585072D-308
CURRENCY	@	8	da -922337203685477.5808 a 922337203685477.5807
STRING	$	32K	Le stringhe possono essere lunghe fino a 32K caratteri (byte)

È bene acquisire un po' di familiarità con questi tipi di dati. Per poter svolgere funzioni avanzate con dati memorizzati in questi formati, bisogna conoscere nei dettagli come sono memorizzati nel BASIC. In particolare, questa conoscenza sarà fondamentale per poter interfacciare il BASIC al linguaggio assembly. Se non si sa come sono organizzati i tipi di dati a livello di byte, non si riuscirà a utilizzarli.
Si passeranno ora in rassegna i diversi formati di dati del BASIC, a partire dal formato più semplice, gli interi.

INTEGER E LONG

I formati INTEGER e LONG sono semplici: gli INTEGER sono memorizzati in una parola e i LONG in due parole. Per esempio, ecco come sono memorizzati alcuni interi:

Valore	*Memorizzato come*
1	&H0001
29	&H001D
32767	&H7FFF

Ecco invece come sono memorizzati alcuni valori interi negativi:

Valore	*Memorizzato come*
-1	&HFFFF
-219	&HFF25
-32768	&H8000

***Nota**: Il bit più significativo, cioè il bit 7, è sempre uguale a 1, negli INTEGER e LONG negativi.*

> ***Consiglio:*** Quello che in BASIC viene chiamato un INTEGER, in linguaggio assembly è una word. Si può predisporre lo spazio in memoria per un INTEGER del BASIC semplicemente usando la direttiva DW.

Questi numeri negativi sono memorizzati nella cosiddetta *notazione in complemento a due*, a cui si è accennato nel Capitolo 3 e che, prima di continuare, si deve approfondire.

NOTAZIONE IN COMPLEMENTO A DUE

Tutta la notazione dei numeri con segno, nei computer, deriva dal semplice fatto che 1 + (-1) = 0. Se su un PC o un PS/2 si vogliono svolgere dei calcoli con i numeri negativi, il numero che si sceglie come -1, quando sommato a 1, deve dare per risultato 0. Sembra una richiesta impossibile da soddisfare. Si supponga di lavorare con valori INTEGER. Esiste un numero a 16 bit che, sommato a 1, dia un risultato 0? Sembrerebbe che il risultato debba essere sempre uguale o maggiore di 1.

In realtà, una risposta c'è. La figura 11.2 mostra che cosa succede. Se si prende &HFFFF, che è 1111111111111111B in binario, e si aggiunge 1, il risultato è 10000000000000000B.

```
 1111111111111111B
             +   1
 10000000000000000B
 ↑
 └──────────────────────────────── Riporto
```

Figura 11.2

Cioè, rimane un INTEGER che contiene 0000000000000000, ovvero 0, e c'è un riporto, poiché l'1 è nella posizione di 2^16 (65536), oltre le possibilità di un intero. Se si ignora il riporto e si considerano solo i 16 bit che possono essere memorizzati in un intero, rimane 00000000000000000B. In altre parole, HFFFF + 1 = 0. Questo è esattamente ciò che succede quando si lavora con la notazione in complemento a due. Si ignora il riporto. Questo significa che &HFFFF è il complemento a 2 di 1 in formato INTEGER. Nel formato LONG, questo numero sarebbe &HFFFFFFFF.
Se si prende un numero come 5, che è 0000000000000101B in formato INTEGER, si invertono tutti i bit ottenendo 1111111111111010B (=&HFFFA) e si sommano i due numeri, si ottiene 1111111111111111B (figura 11.3).

```
   0000000000000101B  ←──  5
+  1111111111111010B       5 con bit invertiti (&HFFFA)
   1111111111111111B  =    &HFFFF
```

Figura 11.3

Se si aggiunge 1 al risultato, si ottiene 0 con un riporto (figura 11.4).

```
   0000000000000101B        5  ←──
+  1111111111111010B        5 con tutti i bit invertiti (= &HFFFA)
+                 1
   10000000000000000B       = 0 con riporto
   ↑
   │
   └──────────────────────────────── Riporto
```

Figura 11.4

Poiché si ignora il riporto, il risultato è 0. Perciò -5, il complemento a due di 5, deve essere uguale a &HFFFA + 1, che è &HFFFB. Questo è il modo in cui si trovano i numeri negativi.
Per esempio, per trovare -1, si parte con +1, che è memorizzato in formato INTEGER come 0000000000000001B (16 bit). Innanzitutto, si invertono tutti i bit, trasformando 0 in 1 e 1 in 0: 1111111111111110B. Poi si aggiunge 1 al risultato, ottenendo 1111111111111111B. Questo è il complemento a due di 1, cioè -1 memorizzato come numero binario 1111111111111111B in formato INTEGER, che è &HFFFF (&HFFFFFFFF nel formato LONG).
Il procedimento è lo stesso per qualsiasi altro numero di cui si voglia invertire il segno. Ora sarà possibile lavorare con INTEGER e LONG, sia positivi che negativi, a livello di bit.

SINGLE E DOUBLE

I numeri in virgola mobile sono notevolmente più complicati. Per esempio, un numero come 10.5 è memorizzato in BASIC come &H41280000: la somiglianza con il valore originale è davvero scarsa. Perché?
Bisogna ricordare sempre che il comuter è una macchina binaria, il che significa che utilizza la base 2 e un numero come 10.5 deve essere memorizzato nella forma 1 x 2^3 più 1 x 2^1 più 1 x 2^-1, che è 1010.1, dove il punto è un punto binario. Qualsiasi numero che possa essere rappresentato nel sistema decimale può essere rappresentato anche nel binario (semplicemente ci vogliono spesso più cifre). Per esempio, 0.75 può essere ottenuto come 1/2 + 1/4, ovvero 2^-1 + 2^-2, perciò 0.75 = .11B.
I numeri memorizzati in virgola mobile sono tutti *normalizzati*, il che significa che vengono sempre rappresentati con la virgola (il punto binario) vicino all'inizio. Per esempio, 1010.1B in forma esponenziale è 1.0101 x 2^3. Per svilupparlo, basta spostare verso destra di tre posizioni il punto binario. In un numero binario normalizzato, la prima cifra è sempre 1.
Nel formato BASIC per la virgola mobile, l'1 iniziale è *implicito*. In altre parole, tutto ciò che si memorizza del *significando* è 0101B. L'esponente è ancora 3, ma, per rendere il tutto ancora un po' più complicato, gli esponenti vengono memorizzati dopo essere stati sommati a un dato offset o *bias*.
Per i SINGLE, questo bias è &H7F (cioè 127); per i DOUBLE è &H3FF (cioè 1023). In altre parole, se il numero 1010.1B deve essere memorizzato come SINGLE il suo esponente sarà 3 + 127 = 130 = &H82.
Infine, il primo bit di qualsiasi numero in virgola mobile è il bit del segno. I numeri in virgola mobile non sono memorizzati in complemento a due. Se il bit del segno è 1, il numero è negativo; se è 0, il numero è positivo.
Questa è l'unica distinzione fra numeri positivi e negativi. Il formato SINGLE (4 byte) è rappresentato in figura 11.5.

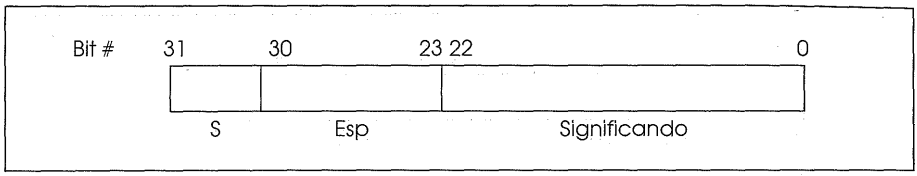

Figura 11.5

Perciò 10.5 = 1010.1B sarà rappresentato come in figura 11.6.

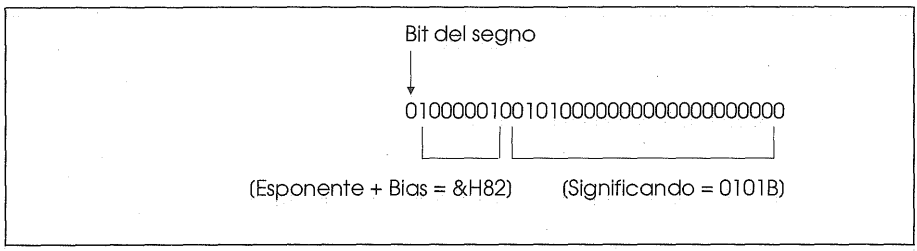

Figura 11.6

Lo si può trasformare in esadecimale raggruppando le cifre binarie a quattro a quattro, poiché quattro cifre binarie rappresentano esattamente una cifra esadecimale (figura 11.7).

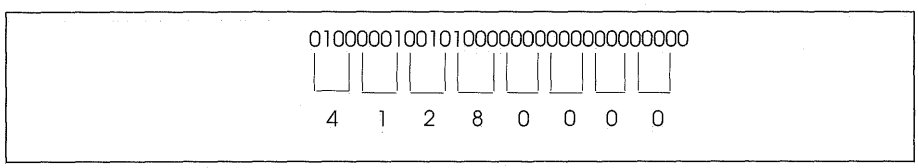

Figura 11.7

Perciò 10.5 è memorizzato come &H41280000.
Il formato per i DOUBLE (8 byte) è mostrato in figura 11.8.

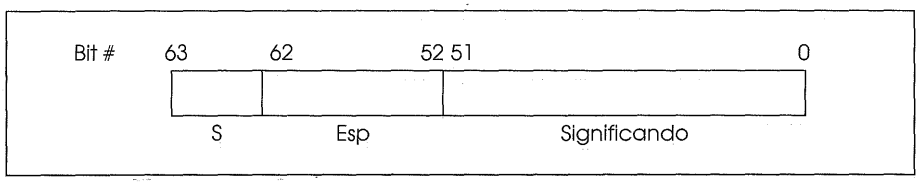

Figura 11.8

Val la pena di notare che il formato dei SINGLE e dei DOUBLE è il formato standard IEEE, che è lo stesso utilizzato dai coprocessori matematici 80x87: l'interfacciamento con questi coprocessori è quindi molto semplice.

> ***Consiglio:*** Il modo più semplice per gestire i numeri in virgola mobile del BASIC dal linguaggio assembly consiste nel lasciare che il BASIC gestisca tutta la matematica in virgola mobile e poi passare sia la mantissa sia l'esponente al codice in linguaggio assembly sotto forma di interi, evitando la necessità di decodificarli.

CURRENCY

Il tipo valuta (CURRENCY) è molto facile. Si tratta semplicemente in un intero di 8 byte in complemento a due, scalato di 10000. Cioè, per trovare la quantità rappresentata da una variabile di tipo CURRENCY basta trattare gli otto byte come un numero unico e dividere per 10000. La parte frazionaria rappresenta i centesimi (trattandosi di quattro cifre, la precisione è di 1/100 di centesimo) e il numero intero rappresenta la cifra. Per esempio, un dollaro sarebbe memorizzato come 10000, cioè &H2710, cioè in questo modo:

```
&H00 &H00 &H00 &H00 &H00 &H00 &H27 &H10
```

Per finire resta un ultimo tipo di dati standard, STRING.

STRINGHE E MATRICI

Stringhe e matrici vengono memorizzate con *descrittori* di stringa; qui si lavorerà tuttavia solo con i descrittori utilizzati da BC.EXE e QB.EXE. I descrittori di stringhe e matrici utilizzati da QBX.EXE sono proprietari e sarà necessario utilizzare delle routine BASIC particolari per utilizzarli in altri modi. Sotto BC.EXE o QB.EXE, un *descrittore di stringa* è organizzato come in figura 11.9.

Qui LEN(Stringa) è la lunghezza della stringa, memorizzata in due byte, e SADD (Stringa) è l'indirizzo dei dati della stringa, a loro volta memorizzati in byte (cioè un indirizzo di offset). Quello che il BASIC passa è l'indirizzo del descrittore, non

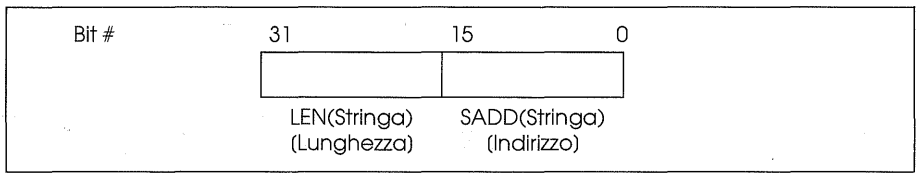

Figura 11.9

l'indirizzo degli effettivi dati della stringa. Per trovare i dati, si deve leggere l'indirizzo di offset della stringa dal descrittore (come si farà più avanti).
Nel QuickBASIC esteso, QBX.EXE, bisogna trovare lunghezza e indirizzo della stringa in altro modo. QBX.EXE ha due routine interne che utilizza per trovare indirizzo e lunghezza di una stringa, denominate rispettivamente StringAddress( ) e StringLength( ). Quando il BASIC passa un descrittore di stringa, bisogna passarlo a queste due routine interne per trovare l'effettiva lunghezza e l'effettivo indirizzo della stringa.
Inoltre, quando si ripassa una stringa a un programma sotto QBX, bisogna costruire un descrittore di stringa che QBX sia in grado di riconoscere: per questo si usa una subroutine interna di QBX denominata StringAssign( ).
I descrittori di matrici sono notevolmente più complessi di quelli per le stringhe. Tuttavia, è utile sapere come il BASIC memorizzi gli elementi di una matrice. Per esempio, anche se si sa l'indirizzo di Array(1, 1), dove si trova Array(5, 1)? Se si vuole riempire una matrice come questa (dal capitolo 7):

```
DIM Array(10, 2) AS CURRENCY

REM Inserisce in Array(n,1) le vendite del giorno:

Array(1, 1) = 10.00
Array(2, 1) = 53.00
Array(3, 1) = 7.00
Array(4, 1) = 9.67
Array(5, 1) = 87.99
Array(6, 1) = 14.00
Array(7, 1) = 91.19
Array(8, 1) = 12.73
Array(9, 1) = 1.03
Array(10, 1) = 5.04

REM Inserisce in Array(n, 2) le vendite del giorno precedente:

Array(1, 2) = 9.67
Array(2, 2) = 3.5
Array(3, 2) = 8.97
Array(4, 2) = 10.00
Array(5, 2) = 78.33
Array(6, 2) = 17.00
Array(7, 2) = 91.36
Array(8, 2) = 12.73
Array(9, 2) = 16.12
Array(10, 2) = 7.98
        :
        :
```

Consiglio: Anziché tentare di codificare i descrittori di matrice, la cosa migliore da fare è semplicemente passare al codice in linguaggio assembly l'indirizzo del primo elemento della matrice.

La figura 11.10 mostra la matrice che viene prodotta.

Col 1	Col 2	
10.00	9.67	← Riga 1
53.00	3.5	← Riga 2
7.17	8.97	
9.67	10.00	
87.99	78.33	
14.00	17.00	
91.19	91.36	
12.73	12.73	
1.03	16.12	
5.04	7.98	

Figura 11.10

Internamente, il BASIC memorizza le matrici di default in *ordine principale di colonna*, il che significa che vengono memorizzati tutti i numeri della colonna 1, poi tutti quelli della colonna 2 e così via.
In memoria quei numeri si trovano in ordine ascendente, in questo modo:

```
Col 1
10.00          Riga 1 (cioè Array(1, 1)
53.00          Riga 2
 7.17  :
 9.67
 87.99
14.00
 91.19
12.73
 1.03
```

```
  5.04
 Col 2
  9.67        Riga 1
  3.5         Riga 2
  8.97        :
 10.00
 78.33
 17.00
 91.36
 12.73
 16.12
  7.98
```

> ***Consiglio:*** Utilizzando l'ordine principale di colonna si può elaborare una matrice in modo da trovare l'elemento desiderato. Per esempio, nel caso di una matrice di interi, ogni elemento è lungo una word. Se si sa dove comincia in memoria la matrice, si può trovare l'elemento desiderato direttamente, invece di fare affidamento sul BASIC.

Ora si può iniziare a scrivere un po' di codice e interfacciarlo al BASIC.

UNA FUNZIONE SENZA PARAMETRI

Il primo esempio è una funzione che non ha parametri e restituisce un intero. Il programma esempio è denominato Return5%( ) e semplicemente restituisce 5 ogniqualvolta viene chiamato. Questa procedura si comporta come una funzione BASIC che restituisca un valore intero 5. Si inizia dichiarando il *modello di memoria* standard per il BASIC. Quando si scrive un programma, non è necessario limitarsi a un solo segmento. In effetti, non è necessario nemmeno limitare a un solo segmento i dati oppure il codice. Tuttavia, se si vogliono usare più segmenti per i dati o per il codice, le etichette devono essere memorizzate dall'assembler in due word, cioè nella forma segmento: offset, e non semplicemente come indirizzo di offset. La Microsoft ha definito vari modelli di memoria che corrispondono alle dimensioni possibili delle sezioni del codice e dei dati di un programma, e il BASIC Microsoft utilizza il modello MEDIUM, il che significa che il codice può occupare più di 64K, ma i dati non possono superare i 64K.

Qui si userà sempre il modello di memoria di default, il modello MEDIUM. Tutto quello che si deve fare è specificare all'assembler questo modello con una direttiva .MODEL. Nella stessa linea di codice, si dice all'assembler che il programma chia-

mante utilizzerà le convenzioni di chiamata BASIC (in altre parole, che i parametri verranno caricati nello stack in ordine di comparsa):

```
.MODEL MEDIUM, BASIC
        :
        :
```

Poi si inizia il segmento del codice con la direttiva .CODE:

```
.MODEL MEDIUM, BASIC
.CODE
        :
        :
```

Poi bisogna dichiarare *PUBLIC* la procedura Return5. Questo significa che l'assemblere inserirà il nome Return5 nell'intestazione del file .OBJ, dove LINK.EXE può trovarlo (senza istruzione PUBLIC, non si potrebbe collegare Return5 al BASIC). Si inizia anche a definire la procedura:

```
.MODEL MEDIUM, BASIC
.CODE
        PUBLIC Return5
Return5 PROC
        :
        :
        RET
Return5 ENDP
        END
```

Infine, è il momento di scrivere la procedura stessa. Si useranno parecchi dei registri, ed è buona pratica di programmazione conservare i contenuti originali di quei registri e ripristinarli poi, nell'eventualità che il BASIC ne abbia bisogno. Lo si può fare salvando i contenuti originali dei registri da utilizzare sullo stack (per poi estrarli di nuovo quando si esce):

```
.MODEL MEDIUM, BASIC
.CODE
        PUBLIC Return5
Return5 PROC
        PUSH BX         ;Bisogna salvare tutti i registri
        PUSH CX
        PUSH ES
        PUSH DS
        ;Qui si svolge l'elaborazione
        :
        RET
Return5 ENDP
        END
```

Poiché si vuole che venga restituito un valore intero 5, bisogna caricare 5 in AX dopo aver ripristinato gli altri registri; infine si ritorna. Il listato 11.2 riporta la funzione completa.

```
.MODEL MEDIUM, BASIC
.CODE
        PUBLIC Return5
Return5 PROC
        PUSH BX         ;Bisogna salvare tutti i registri
        PUSH CX
        PUSH ES
        PUSH DS
        ;Qui si svolge l'elaborazione

        POP  DS         ;Ripristina i registri
        POP  ES
        POP  CX
        POP  BX
        MOV AX,5        ;Restituisce il valore intero 5
        RET
Return5 ENDP
        END
```

Listato 11.2. *La funzione Return5*

Al BASIC Return5%( ) appare esattamente come una normale funzione BASIC. La si dichiara e la si usa in questo modo:

```
DECLARE FUNCTION Return5%( )
PRINT "Il valore restituito da Return5 è: ";Return5%
```

Si può collegare Return5%( ) a questo programma BASIC e poi eseguirlo.

UNA FUNZIONE CON UN PARAMETRO

Poi si può sviluppare un programma che legga un parametro intero e restituisca un altro parametro intero. Per esempio, si è discussa, all'inizio del capitolo, la funzione Add5%( ), che semplicemente sommava 5 all'intero in ingresso e restituiva il risultato. Si può cominciare con la definizione PROC estesa, così:

```
        PUBLIC Add5
Add5    PROC FAR USES DI SI DS ES, VALUE1:WORD
        :
        :
```

Ora si prende da BX l'indirizzo dell'intero che è stato passato:

```
        PUBLIC Add5
Add5    PROC FAR USES DI SI DS ES, VALUE1:WORD

        MOV BX, VALUE1   ;Estrae l'indirizzo di VALUE dallo
                         ;stack (il BASIC passa il riferimento)
        :
        :
```

Il BASIC passa l'indirizzo del parametro, non il parametro effettivo. Si usa quell'indirizzo per ottenere il parametro effettivo, poi gli si aggiunge 5 in questo modo:

```
        PUBLIC Add5
Add5    PROC FAR USES DI SI DS ES, VALUE1:WORD

        MOV BX, VALUE1   ;Estrae l'indirizzo di VALUE dallo
                          ;stack (il BASIC passa il riferimento)
        MOV AX, [BX]
        ADD AX, 5

        RET
Add5    ENDP
        END
```

Poiché il valore da restituire deve essere in AX, tutto è fatto. Il listato 11.3 presenta la funzione completa.

```
 .MODEL MEDIUM, BASIC
 .CODE

        PUBLIC Add5
Add5    PROC FAR USES DI SI DS ES, VALUE1:WORD

        MOV BX, VALUE1   ;Estrae l'indirizzo di VALUE dallo
                         ;stack (il BASIC passa il riferimento)
        MOV AX, [BX]
        ADD AX, 5

        RET
Add5    ENDP
        END
```

Listato 11.3 *La funzione Add5*

Ecco come utilizzare Add5%( ), che, per il BASIC, è una funzione come tutte le altre:

```
REM Esempio di interfacciamento al linguaggio assembly
REM Add5 somma 5 a un intero

DECLARE FUNCTION Add5%(V%)

PRINT "9 + 5 =";Add5%(9%)
```

UNA FUNZIONE CON UN PARAMETRO INTERO LONG

Ora si vedrà come ricevere un intero LONG quale parametro modificando Add5%( ) in Add5Long&( ). Anche se si legge un valore LONG, il suo indirizzo di offset è comunque passato come valore lungo una word. Si carica il LONG nella coppia di registri DX:AX in questo modo:

```
          PUBLIC Add5Long
Add5Long  PROC FAR USES DI SI DS ES, LONG_VALUE:WORD

          MOV BX, LONG_VALUE    ;Estrae dallo stack l'indirizzo
                                ;di LONG_VALUE (il BASIC
          MOV AX, [BX]          ;passa il riferimento)
          MOV DX, [BX+2]
          :
          :
```

Dato che un LONG è memorizzato in due parole, si legge il valore a [BX] e poi anche la parola successiva (cioè quella che si trova 2 byte oltre): [BX + 2]. Questi valori vengono caricati in DX:AX. Ora bisogna sommare 5 a questo intero LONG, cosa che si può fare in questo modo:

```
ADD AX, 5            ;Somma 5 a DX:AX
ADC DX, 0
```

Qui si usa l'istruzione ADD per sommare 5 al valore che si trova in AX, ma questo può produrre un riporto, che si deve sommare alla parola alta in DX. Lo si fa con l'istruzione ADC, cioè *ADD with Carry* (somma con riporto). Si usa semplicemente ADC DX, 0, che somma a DX 0 e l'eventuale riporto generato dall'istruzione precedente. L'articolare le somme con ADD e ADC consente di ottenere risultati con precisione elevata. Ecco come utilizzare queste due istruzioni:

```
          PUBLIC Add5Long
Add5Long  PROC FAR USES DI SI DS ES, LONG_VALUE:WORD

          MOV BX, LONG_VALUE   ;Estrae dallo stack l'indirizzo
                               ;di LONG_VALUE (il BASIC
```

```
        MOV AX, [BX]          ; passa il riferimento)
        MOV DX, [BX+2]
        ADD AX, 5             ;Somma 5 a DX:AX
        ADC DX, 0

        RET
```

***Nota**: Oltre a ADC, esiste anche SBB, Subtract with Borrow (sottrazione con prestito).*

Poiché il LONG è previsto venga restituito in DX:AX, ed è già caricato in questi registri, la codifica è finita. Il listato 11.4 presenta la funzione completa.

```
 .MODEL MEDIUM, BASIC
 .CODE
        ;Estrae un LONG dallo stack

         PUBLIC Add5Long
Add5Long PROC FAR USES DI SI DS ES, LONG_VALUE:WORD

         MOV BX, LONG_VALUE  ;Estrae dallo stack l'indirizzo
                             ;di LONG_VALUE (il BASIC
         MOV AX, [BX]        ; passa il riferimento)
         MOV DX, [BX+2]
         ADD AX, 5           ;Somma 5 a DX:AX
         ADC DX, 0

         RET
Add5Long ENDP
         END
```

Listato 11.4. *La funzione Add5Long*

UNA FUNZIONE CHE RESTITUISCE UNA STRINGA

Ora si può sviluppare un esempio che restituisca al BASIC la stringa "QQQ". Lo si chiamerà GetQQQ$( ). La funzione non prende parametri, ma si limita a restituire una stringa.
QBX mantiene i dati delle stringhe e delle matrici in un segmento speciale, distinto dal resto dei dati, mentre i compilatori BC e QB li conservano nello stesso segmento. Per questo motivo i descritto delle stringhe sono di tipo diverso, e si deve trattare QBX separatamente.

GETQQQ$( ) CON QB.EXE E BC.EXE

In questa versione di GetQQQ$( ), si costruirà un descrittore di stringa da restituire al BASIC, che sarà organizzato come in figura 11.11.

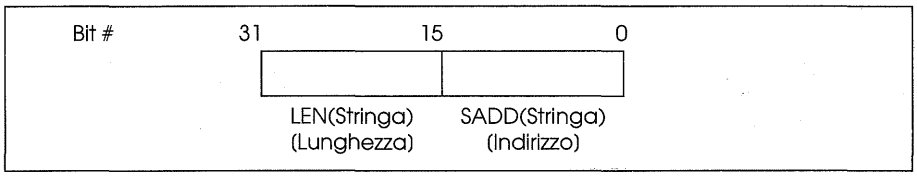

Figura 11.11

Per restituire una stringa, bisogna che il descrittore costruito venga collocato all'indirizzo di offset specificato dal BASIC (e quello stesso indirizzo deve essere restituito in AX).
Quando si riceve il controllo, il BASIC ha già passato in [BP+6] l'indirizzo di offset del valore di ritorno. Si carica quell'indirizzo sia in BX (così da poter utilizzare BX come indice indiretto) sia in AX (perché comunque bisogna restituire quell'indirizzo in AX per il BASIC).

```
        PUBLIC GetQQQ
GetQQQ  PROC FAR USES BX

        MOV     BX, [BP+6]    ;Restituisce un offset valido
        MOV     AX, BX        ;Deve essere restituito anche in AX
        :
        :
```

Ora bisogna impostare il descrittore di stringa a quell'indirizzo. La prima parola sarà 3, lunghezza della stringa "QQQ":

```
        PUBLIC GetQQQ
GetQQQ  PROC FAR USES BX

        MOV     BX, [BP+6]          ;Restituisce un offset valido
        MOV     AX, BX              ;Deve essere restituito anche
                                    ;in AX
        MOV     WORD PTR [BX], 3    ;Lunghezza della stringa
        :
        :
```

L'espressione WORD PTR richiede un'annotazione. Queste parole chiave speciali sono necessarie per una situazione che si presenta solo nel linguaggio assembly: il caso in cui si scambiano dati fra due operandi, nessuno dei quali ha una dimensione definita.

In BASIC, tutte le variabili hanno un tipo assegnato, come INTEGER o LONG, perciò il problema non si presenta mai. Anche nel linguaggio assembly il problema non si presenta nella maggior parte dei casi. Per esempio, non c'è problema con un'istruzione come:

```
MOV AX, 3
```

L'assembler sa che, poiché AX è lungo una word, si intende riempire tutta la parola con il valore 3. Analogamente, se si è definita MIO_NUMERO come MIO_NUMERO DB 5, si può poi cambiare quel numero in 3 in questo modo:

```
MOV MIO_NUMERO, 3
```

Non ci sarebbe problema perché l'assembler sa che la variabile MIO_NUMERO sta per un byte. Invece, se si usasse l'istruzione:

```
MOV [BX], 3
```

l'assembler, che può trattare sia valori di un byte, sia valori di una word, non saprebbe che cosa fare: si vuole riempire la word o il byte puntato da [BX]? In questo caso l'assembler darebbe un errore, perché nessuno dei due operandi ha una dimensione ben definita. La cosa giusta da fare è specificare la dimensione, o con BYTE PTR:

```
MOV BYTE PTR [BX], 3
```

nel qual caso il byte alla locazione [BX] sarebbe riempito con 03H; oppure con WORD PTR:

```
MOV WORD PTR [BX], 3
```

nel qual caso la word in [BX] sarebbe riempita con 0003H.
La successiva word nel descrittore di stringa conterrà l'indirizzo di offset dei dati della stringa. Si memorizzino i dati della stringa immediatamente dopo il descrittore della stringa stesso (cioè bisogna aggiungere 4 byte al valore attuale di BX per saltare oltre il descrittore). Si tratta di un posto comune per memorizzare i dati di stringa in BASIC. Si può puntare al byte subito dopo il descrittore di stringa e collocare anche quell'indirizzo nel descrittore stesso, in questo modo:

```
        PUBLIC GetQQQ
GetQQQ  PROC FAR USES BX

        MOV     BX, [BP+6]          ;Restituisce un offset valido
        MOV     AX, BX              ;Deve essere restituito anche
                                    ;in AX
        MOV     WORD PTR [BX], 3    ;Lunghezza della stringa
        ADD     BX, 4
        MOV     [BX-2], BX          ;Mette l'indirizzo dei dati
                                    ;della stringa nel descrittore
        :
        :
```

La cosa successiva da fare è memorizzare la stringa, "QQQ". Ma, come si è appena visto, non lo si può fare in questo modo:

```
MOV     WORD PTR [BX], 3        ;Lunghezza della stringa
ADD     BX, 4
MOV     [BX-2], BX              ;Mette l'indirizzo dei
                                ;dati della stringa nel
                                ;descrittore
MOV     [BX], "Q"
MOV     [BX+1], "Q"
MOV     [BX+2], "Q"
```

perché "Q" è solo un valore numerico, 51H, per l'assembler, che vede questo:

```
MOV     WORD PTR [BX], 3        ;Lunghezza della stringa
ADD     BX, 4
MOV     [BX-2], BX              ;Indirizzo dei dati
                                ;della stringa
MOV     [BX], 51H
MOV     [BX+1], 51H
MOV     [BX+2], 51H
```

In questo caso, né l'uno né l'altro operando ha una dimensione definita (si intende 51H o 0051H?), perciò bisogna memorizzare la stringa come segue, dopodiché si può ritornare:

```
GetQQQ  PROC FAR USES BX

        MOV     BX, [BP+6]              ;Restituisce un offset
valido
        MOV     AX, BX                  ;Deve essere restituito
anche in AX
        MOV     WORD PTR [BX], 3        ;Lunghezza della stringa
        ADD     BX, 4
        MOV     [BX-2], BX              ;Indirizzo dei dati
                                        ;della stringa
        MOV     BYTE PTR [BX], "Q"      ;Memorizza stringa
        MOV     BYTE PTR [BX+1], "Q"
        MOV     BYTE PTR [BX+2], "Q"

        RET
GetQQQ  ENDP
```

Ed è tutto: la funzione ha restituito una stringa al BASIC sotto BC.EXE o QB.EXE. Il listato 11.5 mostra il programma completo.

```
 .MODEL MEDIUM, BASIC
 .CODE

        PUBLIC GetQQQ
GetQQQ  PROC FAR USES BX

        MOV     BX, [BP+6]          ;Restituisce un offset valido
        MOV     AX, BX              ;Deve essere restituito anche
                                    ;in AX
        MOV     WORD PTR [BX], 3    ;Lunghezza della stringa
        ADD     BX, 4
        MOV     [BX-2], BX                  ;Indirizzo dei dati
                                            ;della stringa
        MOV     BYTE PTR [BX], "Q"          ;Memorizza stringa
        MOV     BYTE PTR [BX+1], "Q"
        MOV     BYTE PTR [BX+2], "Q"

        RET
GetQQQ  ENDP
        END
```

Listato 11.5. *La funzione GetQQQ.*

Ed ecco come si può usare GetQQQ$(), proprio come qualsiasi altra funzione BASIC che restituisca una stringa:

```
REM Esempio di interfaccia per il linguaggio assembly
REM GetQQQ$ restituisce la stringa "QQQ".

DECLARE FUNCTION GetQQQ$( )

PRINT GetQQQ$
```

GETQQQ$( ) CON QBX.EXE

I descrittori di stringa di QBX devono essere realizzati da QBX: non sono nella stessa forma semplice usata da BC.EXE o QB.EXE. Si userà perciò lo stesso sottoprogramma interno che utilizza QBX per costruire i suoi descrittori di stringa. Il sottoprogramma è chiamato StringAssign( ) e si può chiamarlo in questo modo:

```
CALL StringAssign(IndirizzoStringaSorgente,
LungStringaSorgente%, IndirizzoDesc, LungStringaDestinaz%)
```

In questo caso tutti i valori sono passati per valore, non per riferimento. Qui, IndirizzoStringaSorgente è l'indirizzo della stringa per cui si vuole un descrittore; LungStringaSorgente% è la lunghezza di quella stringa; IndirizzoDesc è l'indirizzo a

cui si vuole collocare il descrittore; LunghezzaStringaDestinaz% sarà impostato a 0 (a indicare che si crea una stringa BASIC).
Per cominciare, si impostano i dati nel *segmento dei dati* (poiché il codice eseguibile finale sarà un file .EXE, qui si possono usare sia un segmento di codice sia un segmento di dati). Quando il programma è collegato con il LINK, i dati andranno automaticamente nel segmento di dati del file .EXE completo.
Ecco come si memorizzeranno i dati nel codice che si collega al BASIC: nel segmento dei dati. Bisognerà memorizzare la stringa stessa in memoria, oltre a predisporre lo spazio per un descrittore di stringa lungo due word, perciò si imposta un segmento di dati con la direttiva .DATA:

```
.MODEL MEDIUM, BASIC
.DATA
        Ret_Str         DB      "QQQ"
        Ret_Desc_1      DW      0
        Ret_Desc_0      DW      0
        :
```

Si è definita la stringa di ritorno Ret_Str come "QQQ" e sono state definite due word (mediante la direttiva DW), Ret_Desc_1 e Ret_Desc_2, per contenere il descrittore che darà StringAssign( ). Ora si può iniziare il segmento del codice con .CODE. Come nel caso di BC.EXE o QB.EXE, si salva l'offset del valore di ritorno che viene passato:

```
.MODEL MEDIUM, BASIC
.DATA
        Ret_Str         DB      "QQQ"
        Ret_Desc_1      DW      0
        Ret_Desc_0      DW      0
 .CODE

        PUBLIC GetQQQ
GetQQQ  PROC FAR USES BX

        MOV     BX, [BP+6]      ;Restituisce un offset valido
        MOV     AX, BX          ;Deve essere restituito anche in AX
        :
```

Poi si deve chiamare StringAssign( ) per ottenere un descrittore per "QQQ", cioè Ret_Str, in modo da poter passare di nuovo quel descrittore al programma chiamante come valore di ritorno della funzione. Ecco come può essere formulata la chiamata a StringAssign( ):

```
CALL StringAssign(IndirizzoStringaSorgente,
LungStringaSorgente%, IndirizzoDesc, LungStringaDestinaz%)
```

Per chiamare StringAssign( ), bisogna caricare sullo stack i parametri esattamente come se si trattasse di un programma BASIC. Il primo parametro da passare a StringAssign( ) è l'indirizzo di "QQQ", la stringa che si vuole restituire. Bisogna

caricare (in DS) l'indirizzo del segmento, seguito dal suo offset (OFFSET Ret_Str). Poi bisogna passare la lunghezza della stringa, che è 3:

```
.MODEL MEDIUM, BASIC
.DATA
        Ret_Str         DB      "QQQ"
        Ret_Desc_1      DW      0
        Ret_Desc_0      DW      0
.CODE

        PUBLIC GetQQQ
GetQQQ  PROC FAR USES BX

        MOV     BX, [BP+6]      ;Restituisce un offset valido
        MOV     AX, BX          ;Deve essere restituito anche in AX
        PUSH    DS              ;Indirizzo della stringa QQQ
        MOV     DX, OFFSET Ret_Str
        PUSH    DX
        MOV     DX, 3                   ;Lunghezza della stringa QQQ
        PUSH    DX
        :
```

A questo punto bisogna passare l'indirizzo a cui si vuole che il BASIC scriva il descrittore della stringa. Sono state predisposte a questo proposito due word, Ret_Desc_1 e Ret_Desc_2. Si passa l'indirizzo della prima di queste word, poi una word uguale a 0 per indicare che si vuole che la stringa sia una stringa BASIC. Ora si è pronti a chiamare StringAssign( ):

```
.MODEL MEDIUM, BASIC
.DATA
        Ret_Str         DB      "QQQ"
        Ret_Desc_1      DW      0
        Ret_Desc_0      DW      0

.CODE
        PUBLIC GetQQQ
GetQQQ  PROC FAR USES BX
        MOV     BX, [BP+6]              ;Restituisce un offset valido
        MOV     AX, BX                  ;Deve essere restituito
                                         anche in AX
        PUSH    DS                      ;Indirizzo della stringa QQQ
        MOV     DX, OFFSET Ret_Str
        PUSH    DX
        MOV     DX, 3                   ;Lunghezza della stringa QQQ
        PUSH    DX
        PUSH    DS                      ;Indirizzo del descrittore
                                         da riempire
        MOV     DX, OFFSET Ret_Desc_1
        PUSH    DX
```

```
        MOV     DX, 0                   ;0 per stringa di lunghezza
                                        ;variabile
        PUSH    DX
        EXTRN   StringAssing: PROC
        CALL    StringAssign
        :
```

Si noti che è stata usata l'istruzione EXTRN StringAssign: PROC. Si tratta di un'istruzione simile alla DECLARE del BASIC: EXTRN dice all'assembler che si dovrà collegare in seguito una procedura chiamata StringAssign( ), ma che ancora non si conosce il suo indirizzo. In questo modo, l'assembler lascia nel codice assemblato lo spazio per quell'indirizzo, e il linker colmerà la lacuna. Quando si ritorna da StringAssign( ), le due word del descrittore saranno in Ret_Desc_1 e Ret_Desc_2: questo è il descrittore che si deve passare al BASIC come valore restituito dalla funzione. QBX ha già dato l'offset del valore di ritorno, perciò si deve collocare il descrittore in quella locazione. Il listato 11.6 presenta il programma completo, nella versione per QBX.

```
REM Esempio di interfaccia per il linguaggio assembly
    -- GetQQQ$ restituisce la stringa "QQQ".
DECLARE FUNCTION GetQQQ$( )
PRINT GetQQQ$

.MODEL MEDIUM, BASIC
.DATA
        Ret_Str         DB      "QQQ"
        Ret_Desc_1      DW      0
        Ret_Desc_0      DW      0

.CODE
        PUBLIC GetQQQ
GetQQQ  PROC FAR USES BX

        MOV     BX, [BP+6]          ;Restituisce un offset valido
        MOV     AX, BX              ;Deve essere restituito
                                     anche in AX
        PUSH    DS                  ;Indirizzo della stringa QQQ
        MOV     DX, OFFSET Ret_Str
        PUSH    DX
        MOV     DX, 3               ;Lunghezza della stringa QQQ
        PUSH    DX
        PUSH    DS                  ;Indirizzo del descrittore
                                     da riempire
        MOV     DX, OFFSET Ret_Desc_1
```

continua

Listato 11.6. *Il programma GetQQQ$ (versione per QBX)*

```
        PUSH    DX
        MOV     DX, 0                   ;0 per stringa di lunghezza
                                         variabile
        PUSH    DX
        EXTRN   StringAssing: PROC
        CALL    StringAssign
        MOV     DX, Ret_Desc_1          ;Descrittore di ritorno
        MOV     [BX], DX
        MOV     DX, Ret_Desc_2
        MOV     [BX+2], DX
        RET
GetQQQ  ENDP
        END
```

Listato 11.6. *Il programma GetQQQ$ (versione per QBX)*

Ci vuole un po' di lavoro in più, ma questo è il modo in cui si deve procedere, se si vuole utilizzare QBX.EXE.

UNA FUNZIONE CON DUE PARAMETRI

Ora il programma si può espandere un po'. Si possono leggere dal BASIC due due parametri con la stessa facilità di uno solo. Si scriverà una funzione chiamata Addem%( ), che prende due interi, li somma, e restituisce il risultato. Qui si vogliono leggere due parametri (due interi), e lo si può fare usando PROC in questo modo:

```
        PUBLIC Addem
Addem   PROC FAR USES DI SI DS ES, VALUE1:WORD, VALUE2:WORD
        :
        :
```

Ora VALUE1 e VALUE2 contengono gli indirizzi dei due interi. È possibile caricarli e sommarli in questo modo:

```
        PUBLIC Addem
Addem   PROC FAR USES DI SI DS ES, VALUE1:WORD, VALUE2:WORD
        MOV BX, VALUE1    ;Prende l'indirizzo di VALUE dallo
                          ;stack (il BASIC passa il riferimento)
        MOV AX, [BX]
        MOV BX, VALUE2
        ADD AX, [BX]

        RET
Addem   ENDP
```

Il risultato viene restituito in AX, ed è tutto. Il listato 11.7 presenta il programma completo.

```
 .MODEL MEDIUM, BASIC
 .CODE

        PUBLIC Addem
Addem   PROC FAR USES DI SI DS ES, VALUE1:WORD, VALUE2:WORD

        MOV BX, VALUE1    ;Prende l'indirizzo di VALUE dallo
                          ;stack (il BASIC passa il riferimento)
        MOV AX, [BX]
        MOV BX, VALUE2
        ADD AX, [BX]

        RET
Addem   ENDP
        END
```

Listato 11.7 *ADDEM.ASM: somma due interi.*

Ed ecco un esempio di uso di Addem%( ):

```
REM Esempio di interfaccia per il linguaggio assembly
REM Addem( ) somma due interi

DECLARE FUNCTION Addem%(V1%,V2%)

PRINT "9 + 5 =";Addem%(5,9)
```

UNA FUNZIONE CHE USA IL SEGMENTO DI DATI

Si è visto brevemente come utilizzare un segmento di dati nell'esempio della stringa per QBX. È opportuno dare un'occhiata più da vicino al problema. Si può modificare la procedura Addem%( ), appena scritta, per memorizzare i dati nel segmento dei dati.

Per memorizzare dati, è necessario raggiungere il segmento dei dati, cosa che si può fare con la direttiva .DATA. Non c'è nulla di più facile. Per memorizzare nelle word di memoria VAL1 e VAL2 i due interi che si vogliono sommare, si usa .DATA in questo modo:

```
.MODEL MEDIUM, BASIC

        PUBLIC Addem
.DATA
        VAL1 DW 0
        VAL2 DW 0
        :
        :
```

Ora ci si può riferire nel codice a VAL1 e VAL2 come se fossero normali variabili del linguaggio assembly:

```
.MODEL MEDIUM, BASIC

        PUBLIC Addem
.DATA
        VAL1 DW 0
        VAL2 DW 0

.CODE
Addem   PROC FAR USES DI SI DS ES, VALUE1:WORD, VALUE2:WORD

        MOV BX, VALUE1    ;Prende l'indirizzo di VALUE dallo
                          ;stack (il BASIC passa il riferimento)
        MOV CX, [BX]
        MOV VAL1, CX
        MOV BX, VALUE2
        MOV CX, [BX]
        MOV VAL2, CX

        ;Ora i dati sono nel gruppo dei dati

        MOV AX, VAL1
        ADD AX, VAL2

        RET
Addem   ENDP
        END
```

Ed è tutto. In questo modo si possono memorizzare e recuperare dati nel segmento dei dati.
Il listato 11.8 presenta il programma completo.

```
 .MODEL MEDIUM, BASIC

        PUBLIC Addem
 .DATA
        VAL1 DW 0
        VAL2 DW 0

 .CODE
Addem   PROC FAR USES DI SI DS ES, VALUE1:WORD, VALUE2:WORD

        MOV BX, VALUE1    ;Prende l'indirizzo di VALUE dallo
                          ;stack (il BASIC passa il riferimento)
        MOV CX, [BX]
        MOV VAL1, CX
        MOV BX, VALUE2
        MOV CX, [BX]
        MOV VAL2, CX

        ;Ora i dati sono nel gruppo dei dati

        MOV AX, VAL1
        ADD AX, VAL2

        RET
Addem   ENDP
        END
```

Listato 11.8. *ADDEM.ASM: una versione che utilizza il segmento dei dati.*

UN SOTTOPROGRAMMA CON PARAMETRI

Ora si svilupperà una versione in linguaggio assembly di un sottoprogramma (non una funzione!) che prende un parametro. Il programma si chiamerà PrintChar( ), prenderà il codice ASCII di un carattere e lo stamperà.
Dal punto di vista del linguaggio assembly, un sottoprogramma BASIC è esattamente come una funzione BASIC, salvo che non è necessario caricare in AX (ed eventualmente anche in DX) con un valore di ritorno.

```
PrintChar PROC FAR USES DI SI DS ES, The_Char:WORD

        MOV BX, The_Char    ;Prende l'indirizzo di VALUE dallo
                            ;stack (il BASIC passa il riferimento)
        MOV DX, [BX]        ;Carica in DL
```

```
        MOV AH, 2
        INT 21H
```

Il listato 11.9 presenta il programma completo.

```
 .MODEL MEDIUM, BASIC
 .CODE

        PUBLIC PrintChar
PrintChar PROC FAR USES DI SI DS ES, The_Char:WORD

        MOV BX, The_Char    ;Prende l'indirizzo di VALUE dallo
                            ;stack (il BASIC passa il riferimento)
        MOV DX, [BX]        ;Carica in DL
        MOV AH, 2
        INT 21H

        RET
PrintChar ENDP
        END
```

Listato 11.9 *PRINTCHAR.ASM*

Si può usare PrintChar( ) in questo modo:

```
REM Esempio di interfaccia del linguaggio assembly
REM PrintChar prende un carattere e lo stampa

DECLARE SUB PrintChar(V%)

CALL PrintChar(ASC("a"))
```

Come nel caso delle funzioni, è possibile che i parametri siano due. Ecco un rapido esempio di un sottoprogramma, chiamato PrintSum%( ), che accetta due interi e stampa il risultato della loro somma (purché il risultato sia una singola cifra, fra 0 e 9):

```
 .MODEL MEDIUM, BASIC
 .CODE

        PUBLIC PrintSum
PrintSum PROC FAR USES DI SI DS ES, VALUE1:WORD, VALUE2:WORD

        MOV BX, VALUE1      ;Prende l'indirizzo di VALUE dallo
                            ;stack (il BASIC passa il riferimento)
        MOV DX, [BX]
        MOV BX, VALUE2
        ADD DX, [BX]
        ADD DX, "0"
```

```
        MOV AH, 2
        INT 21H

        RET
PrintSum ENDP
        END
```

Si leggono i due parametri, che sono due interi, si converte il risultato (che deve essere compreso fra 0 e 9) in un carattere ASCII e infine lo si stampa con il servizio 2 dell'interrupt &H21.

UN SOTTOPROGRAMMA CON UN PARAMETRO STRINGA

Si può sviluppare un sottoprogramma che legga una stringa passata dal BASIC (al contrario di quel che fa GetQQQ$( ), la funzione che restituiva una stringa al BASIC). Per esempio, si può sviluppare un sottoprogramma PrintString( ) che stampi una stringa passatagli come argomento, in questo modo:

```
REM Esempio di interfaccia del linguaggio assembly
REM PrintString riceve una stringa e la stampa

DECLARE SUB PrintString(Strg$)

CALL PrintString("Questa è una prova")
```

> ***Consiglio:*** La gestione delle stringhe è uno degli usi più comuni per il linguaggio assembly. I microprocessori 80x86 hanno numerose istruzioni stringa interne con nomi come MOVS, CMPS e SCAS, che sono molto più veloci delle corrispondenti istruzioni in BASIC.

Anche in questo caso, poiché QBX.EXE gestisce i descrittori di stringa in modo diverso, bisogna formulare l'esempio in due versioni, una per BC.EXE e QB.EXE e l'altra per QBX.EXE.

PRINTSTRING( ) CON BC.EXE E QB.EXE

Si comincerà esaminando il procedimento con BC.EXE e QB.EXE. In questo caso, il programma chiamante passa l'indirizzo del descrittore di stringa quando effettua la chiamata CALL PrintString("Questa è una prova.") Per decodificare il descrittore, si

carica la lunghezza della stringa (la prima word nel descrittore di stringa) in CX e l'indirizzo dei dati della stringa (la seconda word nel descrittore) in BX, in questo modo:

```
PrintString PROC FAR USES DI SI DS ES, The_String:WORD

        MOV AH, 2
        MOV BX, The_String    ;Indirizzo di lung_str:word,
                              ;indir_dati_str:word
        MOV CX, [BX]          ;Lunghezza della stringa
        MOV BX, [BX+2]        ;Indirizzo dei dati della stringa
Print_Loop:
        :
```

Ora è sufficiente percorrere in ciclo i dati della stringa, caricando ciascun byte in DL e stampandolo con il servizio 2, in questo modo:

```
PrintString PROC FAR USES DI SI DS ES, The_String:WORD

        MOV AH, 2
        MOV BX, The_String    ;Indirizzo di lung_str:word,
                              ;indir_dati_str:word
        MOV CX, [BX]          ;Lunghezza della stringa
        MOV BX, [BX+2]        ;Indirizzo dei dati della stringa
Print_Loop:
        MOV DX, [BX]
        INT 21H
        INC BX
        LOOP Print_Loop

            RET
PrintString ENDP
            END
```

Ed è tutto. Il listato 11.10 presenta il programma completo.

```
.MODEL MEDIUM, BASIC
.CODE

        PUBLIC PrintString
PrintString PROC FAR USES DI SI DS ES, The_String:WORD

        MOV AH, 2
        MOV BX, The_String    ;Indirizzo di lung_str:word,
                              ;indir_dati_str:word
        MOV CX, [BX]          ;Lunghezza della stringa
```

continua

Listato 11.10. *PRINTSTRING.ASM*

```
        MOV BX, [BX+2]        ;Indirizzo dei dati della stringa
Print_Loop:
        MOV DX, [BX]
        INT 21H
        INC BX
        LOOP Print_Loop

            RET
PrintString ENDP
            END
```

Listato 11.10. *PRINTSTRING.ASM.*

PRINTSTRING( ) CON QBX.EXE

È un po' più complicato con QBX. Anziché poter leggere lunghezza e indirizzo del descrittore di stringa, bisogna usare due funzioni interne a QBX per trovare quelle informazioni. Queste funzioni sono StringLength(Descrittore) e StringAddress(Descrittore) e l'argomento che si passa è il descrittore di stringa che si vuole decodificare.
Si inizierà il segmento del codice utilizzando StringLength( ) per trovare la lunghezza della stringa che è stata passata. Bisogna tenere ben presente che ciò che è stato passato non è il descrittore della stringa, ma l'indirizzo del descrittore della stringa. Va benissimo, comunque, perché si vuole in effetti passare quell'indirizzo a StringLength( ) e StringAddress( ). Si comincia con il chiamare StringLength( ) con l'indirizzo del descrittore di stringa appena ricevuto dal programma chiamante:

```
.MODEL MEDIUM, BASIC

        PUBLIC PrintString

.CODE
PrintString PROC FAR USES DI SI DS ES, The_String:WORD
        PUSH  The_String
        EXTRN StringLength: PROC
        CALL  StringLength
        MOV   CX, AX        ;Carica la lunghezza della stringa
                            ;in CX per LOOP
        :
```

La lunghezza della stringa è restituita in AX. Come nel caso della precedente versione di PrintString( ), si sposta quel valore in CX per poter usare l'istruzione LOOP per stampare la stringa.
Poi bisogna passare l'indirizzo del descrittore a StringAddress( ). Questa funzione restituisce l'indirizzo (due word) dei dati della stringa. Poiché il valore di ritorno è

lungo due word, DX conterrà la word più significativa (indirizzo di segmento) e AX quella meno significativa (indirizzo di offset):

```
.MODEL MEDIUM, BASIC
        PUBLIC PrintString

.CODE
PrintString PROC FAR USES DI SI DS ES, The_String:WORD
        PUSH  The_String
        EXTRN StringLength: PROC
        CALL  StringLength
        MOV   CX, AX        ;Carica la lunghezza della stringa
                            ;in CX per LOOP

        PUSH  The_String
        EXTRN StringAddress: PROC
        CALL  StringAddress
        :
```

Ora si vuole leggere la stringa, carattere per carattere. Per questo bisogna prima fare sì che l'indirizzo di segmento venga restituito da StringAddress( ) nel segmento dei dati caricandolo in DS. Poi si può caricare in BX l'indirizzo di offset della stringa e leggere i caratteri in questo modo: MOV DL, [BX]. (Come si è visto nel capitolo 10, nel metodo di indirizzamento [BX], BX contiene l'offset rispetto all'inizio del segmento dei dati.)
L'indirizzo di segmento della stringa è in DX e il suo offset è in AX, perciò prima si salva l'indirizzo di segmento originale dei dati e si possono caricare DX e BX in questo modo:

```
.MODEL MEDIUM, BASIC

        PUBLIC PrintString

.CODE
PrintString PROC FAR USES DI SI DS ES, The_String:WORD
        PUSH  The_String
        EXTRN StringLength: PROC
        CALL  StringLength
        MOV   CX, AX        ;Carica la lunghezza della stringa
                            ;in CX per LOOP

        PUSH  The_String
        EXTRN StringAddress: PROC
        CALL  StringAddress
        PUSH  DS                    ;Salva  DS
        MOV   DS, DX                ;DS     Segmento della stringa
        MOV   BX, AX                ;BX     Offset della stringa
        :
```

Ora si è liberi di usare lo stesso ciclo di stampa sviluppato per la precedente versione di PrintString:

```
.MODEL MEDIUM, BASIC

        PUBLIC PrintString

.CODE
PrintString PROC FAR USES DI SI DS ES, The_String:WORD
        PUSH  The_String
        EXTRN StringLength: PROC
        CALL  StringLength
        MOV   CX, AX        ;Carica la lunghezza della stringa
                            ;in CX per LOOP

        PUSH  The_String
        EXTRN StringAddress: PROC
        CALL  StringAddress
        PUSH  DS                    ;Salva  DS
        MOV   DS, DX                ;DS     Segmento della stringa
        MOV   BX, AX                ;BX     Offset della stringa
        MOV AH, 2
Print_Loop:
        MOV DL, [BX]
        INT 21H
        INC BX
        LOOP Print_Loop
        :
```

Alla fine, bisogna ripristinare il valore originale di DS, poi si può ritornare. Il listato 11.11 mostra la versione completa di PrintString per QBX.EXE.

```
.MODEL MEDIUM, BASIC
        PUBLIC PrintString

.CODE
PrintString PROC FAR USES DI SI DS ES, The_String:WORD
        PUSH  The_String
        EXTRN StringLength: PROC
        CALL  StringLength
        MOV   CX, AX        ;Carica la lunghezza della stringa
                            in CX per LOOP

        PUSH  The_String
        EXTRN StringAddress: PROC
        CALL  StringAddress
```

continua

Listato 11.11. *PRINTSTRING.ASM (versione per QBX.EXE).*

```
        PUSH  DS                    ;Salva  DS
        MOV   DS, DX                ;DS     Segmento della stringa
        MOV   BX, AX                ;BX     Offset della stringa

        MOV AH, 2
Print_Loop:
        MOV DL, [BX]
        INT 21H
        INC BX
        LOOP Print_Loop

        POP  DS                     ;Ripristina DS

        RET
PrintString ENDP
        END
```

Listato 11.11. *PRINTSTRING.ASM (versione per QBX.EXE).*

Anche in questo caso, per utilizzare la stringa sotto QBX.EXE è richiesto un po' più di lavoro, ma d'altra parte, se si vuole usare quel compilatore, questa è l'unica possibilità.

UN SOTTOPROGRAMMA CON UN PARAMETRO MATRICE

Come ultimo esempio, si vedrà come passare matrici a un sottoprogramma. Come si è già visto, lavorare con i descrittori di matrici non è facile. È di gran lunga meglio passare l'indirizzo effettivo del primo elemento della matrice.
Anche per le matrici QBX richiede un comportamento diverso, ma questa volta QB.EXE e QBX.EXE lavorano nello stesso modo: con ambedue le matrici possono esse memorizzate in qualsiasi punto della memoria, perciò è necessario passare alle routine in linguaggio assembly sia l'indirizzo di segmento sia quello di offset del primo elemento della matrice. In BC.EXE, le matrici sono semplicemente memorizzate nel segmento dei dati, perciò sarà necessario passero solo un indirizzo lungo una word (l'indirizzo del segmento dei dati è già in DS).
Come esempio, si scriverà un sottoprogramma in grado di stampare il primo elemento di una matrice: lo si chiamerà PrintFirstElement( ).

PRINTFIRSTELEMENT CON BC.EXE

Bisogna passare a PrintFirstElement( ) l'indirizzo del primo elemento di una matrice, che si suppone si chiami A( ). Per questo, non si vuole semplicemente passare VARPTR(A(1)), perché in questo modo si passerebbe l'indirizzo dell'indirizzo di A(1), dato che gli argomenti vengono passati per riferimento. Per evitarlo, si userà nella dichiarazione di PrintFirstElement( ) BYVAL, in modo che il programma passi i parametri per valore e non per riferimento. Poi si è pronti a chiamare PrintFirstElement( ) con un argomento VARPTR(A(1)):

```
REM Esempio di PrintFirstElement
REM Le matrici hanno descrittori complessi -- meglio passare
REM l'indirizzo del primo elemento
REM Si noti l'uso di BYVAL nella dichiarazione SUB

DECLARE SUB PrintFirstElement (BYVAL Addr AS INTEGER)
DIM A(1 TO 6) AS INTEGER

A(1) = 1
A(2) = 2
A(3) = 3
A(4) = 4
A(5) = 5
A(6) = 6

PRINT "A(1) = ";
CALL PrintFirstElement(VARPTR(A(1)))
```

Ora bisogna scrivere la procedura in linguaggio assembly PrintFirstElement( ). Si riceverà l'indirizzo del primo elemento della matrice e tutto quello che si deve fare caricare quell'elemento in DL e stamparlo (dando per scontato che si trovi nell'intervallo fra 0 e 9):

```
PrintFirstElement PROC FAR USES DI SI DS ES,
Array_Seg:WORD,Array_Offset:WORD

        MOV  BX, The_Array_Addr
        MOV  DX, [BX]
        ADD  DX, "0"
        MOV  AH,2
        INT  21H
```

Il listato 11.12 mostra l'intera procedura (versione per BC.EXE).

```
 .MODEL MEDIUM, BASIC
 .CODE

        PUBLIC PrintFirstElement
PrintFirstElement PROC FAR USES DI SI DS ES,
Array_Seg:WORD,Array_Offset:WORD

        MOV  BX, The_Array_Addr
        MOV  DX, [BX]
        ADD  DX, "0"
        MOV  AH,2
        INT  21H

        RET
PrintFirstElement ENDP
        END
```

Listato 11.12. *PrintFirstElement (versione per BC.EXE).*

PRINTFIRSTELEMENT CON QB.EXE E QBX.EXE

Con QBX.EXE e QB.EXE le matrici possono essere memorizzate in qualunque punto della memoria, perciò bisogna passare a PrintFirstElement( ) sia l'indirizzo del segmento, sia l'indirizzo di offset dell'elemento:

```
REM Esempio di PrintFirstElement con QBX
REM Le matrici hanno descrittori complessi -- meglio passare
REM l'indirizzo del primo elemento
REM Si noti l'uso di BYVAL nella dichiarazione SUB

DECLARE SUB PrintFirstElement (BYVAL Addr AS INTEGER, BYVAL
                               Addr2 AS INTEGER)
DIM A(1 TO 6) AS INTEGER
A(1) = 1
A(2) = 2
A(3) = 3
A(4) = 4
A(5) = 5
A(6) = 6
PRINT "A(1) = ";
CALL PrintFirstElement(VARSEG(A(1), VARPTR(A(1)))
```

Poi, nella procedura PrintFirstElement, si colloca l'indirizzo di segmento in DS (dopo aver salvato sullo stack il valore originale), l'indirizzo di offset in BX, e si legge il

primo elemento della matrice in questo modo: MOV DX, [BX]. Si può aggiungere uno "0" ASCII a questo valore per convertirlo in una cifra ASCII (dando per scontato che il suo valore sia compreso fra 0 e 9) e poi lo si stampa. Il listato 11.13 mostra la versione per QB.EXE e QBX.EXE.

```
 .MODEL MEDIUM, BASIC
 .CODE

        PUBLIC PrintFirstElement
PrintFirstElement PROC FAR USES DI SI DS ES,
Array_Seg:WORD,Array_Offset:WORD

        PUSH DS
        MOV  BX, Array_Offset
        PUSH Array_Seg
        POP  DS
        MOV  DX, [BX]
        POP  DS
        ADD  DX, "0"
        MOV  AH,2
        INT  21H

        RET
PrintFirstElement ENDP
        END
```

Listato 11.13. *PrintFirstElement (versione per QB.EXE e QBX.EXE)*

Questo è tutto, ed è anche la fine dell'ultimo esempio in linguaggio assembly.

CONCLUSIONI

Si è fatta molta strada, in queste pagine: si è esaminato ampiamente il computer, passando dal controllo dello schermo ai menu, dalla tastiera al mouse, da un programma originale di disegno a un programma di database. Si è acquisita dell'esperienza nell'interfacciamento del BASIC verso il linguaggio assembly, nell'ordinamento di dati, nel disegno di finestre. Non è mancata nemmeno una rassegna del sistema operativo, e le conoscenze acquisite sono state subito messe all'opera.
In effetti, si può dire che così si è diventati un po' esperti in tutte le aree del BASIC. Si è visto come i programmi professionali ottengano i loro risultati e si è avuta la possibilità di dare una sbirciatina dietro le quinte. L'unica cosa che resta, ora, è sfruttare tutte queste conoscenze. Buona fortuna, e felice programmazione!

APPENDICE

REFERENCE BIOS E DOS

Lo scopo di questa appendice è quello di servire da riferimento. Verranno presentati gli interrupt disponibili che vanno da *0* a *FFH*, prestando più attenzione a quelli che risulteranno di particolare utilità.

INTERRUPT DEL BIOS

Interrupt 0 – Divisione per 0 (errore nella divisione)

È questo il primo degli interrupt BIOS. Il BIOS utilizza gli interrupt che vanno da *0* a *1FH*, mentre il DOS utilizza quelli da *20H* in avanti. L'interrupt 0 è una routine che controlla la divisione per 0 e, se questa si verifica, esso viene immediatamente richiamato generando un mesaggio del tipo "Errore di divisione" e interrompendo, di solito, l'esecuzione del programma.

Interrupt 1 – Passo singolo

Questo interrupt viene utilizzato unicamente dal debugger. Viene utilizzato per esaminare un codice passo-a-passo e viene richiamato tra un'istruzione e la successiva.

Interrupt 2 – Interrupt non mascherabile (NMI)

Si tratta di un interrupt hardware e non potrà essere bloccato utilizzando STI e CLI. Verrà sempre eseguito ogni volta che viene richiamato.

Interrupt 3 – Breakpoint

Si tratta di un altro interrupt utilizzato dal debugger. DEBUG utilizza questo interrupt in associazione con il comando GO. Se si desidera eseguire il codice

fino a un particolare indirizzo e poi interromperne l'esecuzione, DEBUG inserirà un INT 3 al punto di interruzione desiderato e restituirà il controllo al peogramma. Quando verrà raggiunto INT 3, DEBUG potrà nuovamente assumere il controllo.

Interrupt 4 – Overflow

Interrupt molto simile a INT 0. se si verifica una condizione di overflow, viene richiamato questo interrupt. Di solito, non viene richiamata alcuna altra azione e il BIOS restituisce semplicemente il controllo.

Interrupt 5 – Copia su carta dello schermo (Print screen)

Questo interrupt viene selezionato dal BIOS per emettere una copia su carta del contenuto dello schermo. Se viene premuto il tasto STAMP (PrtSc), viene automaticamente richiamato questo interrupt. È inutile dire che un programma potrà richiamare INT 5 semplicemente immettendone al suo interno l'appropriata istruzione. Non viene trasferito alcun tipo di argomento.

Interrupt 6 e 7 – Riservati

Interrupt 8 – Data corrente

Si tratta di un altro interrupt hardware; esso viene richiamato per aggiornare la data corrrente di sistema (memorizzata nell'area dati del BIOS) a una frequenza di circa 18,2 volte al secondo. Se la data dovrà essere modificata, questo interrupt potrà anche gestire questa operazione.
Questo interrupt richiama anche INT 1CH e se si desidera intercettare il timer ed eseguire una qualche operazione alla velocità di 18,2 volte al secondo è consigliabile intercettare INT 1CH invece dell'Interrupt 8.

Interrupt 9 – Tastiera

Interrupt hardware che potrà essere intercettato dai programmi residenti in memoria.

Interrupt 0AH – Riservato

Interrupt da 0BH a 0FH

Questi interrupt puntano alla routine BIOS D_EOI che corrisponde alla routine *End of Interrupt* del BIOS. L'azione di queste routine è quella di reimpostare il gestore dell'interrupt a 20H e restituire il controllo.

INT 10H Servizio 0 – Imposta la modalità video

Input
AH = 0
AL = Modo video

Modo video (in AL)	Righe a video	Numero colori	Scheda video	N. max pagine
0	40x25	Testo B/N	CGA, EGA, VGA	8
1	40x25	Testo colore	CGA, EGA, VGA	8
2	80x25	Testo B/N	CGA, EGA, VGA	4(CGA), 8(EGA, VGA)
3	80x25	Testo colore	CGA, EGA, VGA	4(CGA), 8(EGA, VGA)
4	320x200	4	CGA, EGA, VGA	1
5	320x200	B/N	CGA, EGA, VGA	1
6	640x200	2 (att./disatt.)	CGA, EGA, VGA	1
7	80x25	Monocromatico	MDA, EGA, VGA	1(MDA), 8(EGA, VGA)
8	160x200	16	PCjr	1
9	320x200	16	PCjr	1
AH	640x200	1	PCjr	1
BH	Riservato per utilizzo futuro			
CH	Riservato per utilizzo futuro			
DH	320x200	16	EGA, VGA	8
EH	640x200	16	EGA, VGA	4
FH	640x350	Monocromatico	EGA, VGA	2
10H	640x350	16	EGA, VGA	2
11H	640x480	2	VGA	1
12H	640x480	16	VGA	1
13H	320x200	256	VGA	1

INT 10H Servizio 1 – Imposta tipo del cursore

Input
AH = 1
CH = Linea di scansione iniziale
CL = Linea di scansione finale.

Output
Nuovo cursore.

INT 10H Servizio 2 – Fissa posizione cursore

Input
DH,DL = Riga, Colonna
BH = Numero pagina video
AH = 2

Output
Posizione cursore modificata.

***Nota**: DH,DL = 0,0 = Angolo superiore sinistro.*

INT 10H Servizio 3 – Legge posizione cursore

Input
BH = Numero pagina video
AH = 3

Output
DH,DL = Riga, Colonna del cursore
CH,CL = Modalità attiva del cursore

INT 10H Servizio 4 – Legge posizione penna ottica

Input
AH = 4

Output

AH = 0 →	Penna ottica non attiva
AH = 1	DH,DL = Riga, Colonna posizione penna ottica
	CH = Pixel riga (verticale) da 0 a 199
	BX = Pixel colonna (orizzontale) da 0 a 319.639

INT 10H Servizio 5 – Fissa pagina video attiva

Input
AL = 0-7 (Modalità video 0,1)
AL = 0-3 (Modalità video 2,3)
AH = 5

Output
Modifica della pagina video attiva.

***Nota**: È possibile avere a disposizione diverse pagine solo in modalità alfanumerica (adattatori grafici).*

INT 10H Servizio 6 – Scorre verso l'alto la pagina attiva

Input
AL = # N. di righe bianche al piede (0 → svuota completamente l'area)
CH,CL = Riga, Colonna superiore sinistra dell'area da far scorrere
DH,DL = Riga, Colonna inferiore destra dell'area da far scorrere
BH = Attributo di riempimento delle righe bianche
AH = 6

INT 10H Servizio 7 – Scorre verso il basso la pagina attiva

Input
AL = # N. di righe bianche al piede (0 → svuota completamente l'area)
CH,CL = Riga, Colonna superiore sinistra dell'area da far scorrere
DH,DL = Riga, Colonna inferiore destra dell'area da far scorrere
BH = Attributo di riempimento delle righe bianche
AH = 7

INT 10H Servizio 8 – Legge carattere e attributo della posizione del cursore

Input
BH = Numero pagina attiva
AH = 8

Output
AL = Carattere (ASCII) letto
AH = Attributo del carattere (solo alfanumerici)

INT 10H Servizio 9 – Scrive carattere e attributo della posizione del cursore

Input
BH = Numero pagina attiva
BL: → Modalità testo = Attributo
Modalità grafica = Colore
CX = Conteggio dei caratteri da scrivere
AL = Codice ASCII IBM
AH = 9

Output
Carattere scritto a video alla posizione del cursore

INT 10H Servizio A – Scrive SOLO il carattere alla posizione del cursore

Input
BH = Numero pagina attiva
CX = Conteggio caratteri da scrivere
AL = Codice ASCII IBM
AH = 0AH

Output
Carattere scritto a video alla posizione del cursore

INT 10H Servizio B – Imposta la palette dei colori

Input

BH = ID della palette dei colori

BL BH = 0 → BL = Colore sfondo
BH = 1 → BL = Numero palette colori (0 = Verde/Rosso/Giallo); (1 = Ciano/Magenta/Bianco)

AH = 11

INT 10H Servizio C – Scrive un pixel

Input

DX = Numero riga (0-199)
CX = Numero colonna (0-319.639)
AL = Valore del colore (0-3)
AH = 12

***Nota**: [0,0] corrisponde all'angolo superiore sinistro. Se il bit 7 di AL è pari a 1, il valore del colore sarà soggetto a un'operazione di tipo XOR con il valore attivo del colore del pixel.*

INT 10H Servizio D – Legge pixel

Input

DX = Numero di riga (0-199)
CX = Numero di colonna (0-319.639)
AH = 13

Output

AL = Valore del colore (0-3)

***Nota**: [0,0] corrisponde all'angolo superiore sinistro. Se il bit 7 di AL è pari a 1, il valore del colore sarà sogetto a un'operazione di tipo XOR con il valore attivo del colore del pixel.*

INT 10H Servizio E – Scrive un carattere in modo TTY

Input

AL = Codice ASCII IBM
BL = Colore di sfondo (modalità grafica)
AH = 14

INT 10H Servizio FH – Restituisce la modalità video corrente

Input

AH = 15

Output

AH = Numero delle colonne alfanumeriche a video

AL = Modalità attiva (si veda INT 10H Servizio 0)
BH = Pagina video attiva

INT 10H Servizio 10H – Imposta registri della palette dei colori

Palette di colori di default (0-15) con una scheda EGA.

Valore	*Colore*	*rgbRGB*
0	Nero	000000
1	Blu	000001
2	Verde	000010
3	Ciano	000011
4	Rosso	000100
5	Magenta	000101
6	Marrone	010100
7	Bianco	000111
8	Grigio scuro	111000
9	Blu chiaro	111001
10	Verde chiaro	111010
11	Ciano chiaro	111011
12	Rosso chiaro	111100
13	Magenta chiaro	111101
14	Giallo	111110
15	Bianco intenso	111111

INT 10H Servizio 10H Funzione 0 – Imposta un registro della palette di colori

Input
AH = 10H
AL = 0
BL = Registro gamma colori da impostare (0-15)
BH = Valore da impostare (0-63)

INT 10H Servizio 10H Funzione 1 – Imposta il registro del bordo

Input
AH = 10H
BH = Valore da impostare (0-63)

INT 10H Servizio 10H Funzione 2 – Imposta tutti i registri della palette di colori

Input
AH = 10H
AL = 2
ES:BX = Indirizzo di una tabella a 17 byte contenente le sezioni del colore (0-63)
I byte da 0 a 15 contengono le sezioni dei colori per i registri (0-15) della gamma dei colori
Il byte 16 contiene il nuovo colore del bordo

INT 10H Servizio 10H Funzione 7 – Legge una palette dei colori

Input
AH = 10H
AL = 7
BL = Registro da leggere (valore del colore)

Output
BH = Impostazione del registro

INT 10H Servizio 10H Funzione 8 – Legge il registro del colore del bordo

Input
AH = 10H
AL = 8

Output
BH = Impostazione bordo

INT 10H Servizio 10H Funzione 10H – Imposta il registro DAC

Input
AH = 10H
AL = 10H
BX = Registro da impostare (0-255)
CH = Intensità del verde
CL = Intensità del blu
DH = Intensità del rosso

INT 10H Servizio 10H Funzione 12H – Imposta un blocco di registri DAC

Input
AH = 10H
AL = 12H

BX = Primo registro da impostare (0-255)
CX = Numero dei registri da impostare (1-256)
ES:DX = Indirizzo di una tabella di intensità dei colori.
Per ogni registro DAC vengono utilizzati tre byte (per ogni byte verranno usati solo i 6 bit di ordine inferiore). La tabella viene così impostata: rosso, verde, blu, rosso, verde, blu, ...

INT 10H Servizio 10H Funzione 13H – Seleziona la modalità di colore delle pagine

Input
AH = 10H
AL = 13H
BL = 0 Seleziona modalità di colore delle pagine
BH = 0 Seleziona 4 pagine di registri DAC di 64 registri ciascuno
BH = 1 Seleziona 16 pagine di registri DAC di 16 registri ciascuno
BL = 1 Seleziona il colore attivo della pagina
Per uso con modalità a 4 pagine:
BH = 0 Seleziona il primo blocco di 64 registri DAC
BH = 1 Seleziona il secondo blocco di 64 registri DAC
BH = 2 Seleziona il terzo blocco di 64 registri DAC
BH = 3 Seleziona il quarto blocco di 64 registri DAC
Per uso con modalità a 16 pagine:
BH = 0 Seleziona il primo blocco di 16 registri DAC
BH = 1 Seleziona il secondo blocco di 16 registri DAC
..
..
BH = 14 Seleziona il 15° blocco di 16 registri DAC
BH = 15 Seleziona il 16° blocco di 16 registri DAC

INT 10H Servizio 11H – Generatore di caratteri

INT 10H Servizio 12H – Selezione alternativa

Input
AH = 12H
BL = 30H
AL = 0 → 200 linee di scansione dello schermo
= 1 350 linee di scansione dello schermo
= 2 400 linee di scansione dello schermo

INT 11H – Determinazione della configurazione

Output
Bit di AX
15,14 = Numero delle stampanti
13 Non utilizzato

12	Adattatore giochi collegato
11,10,9	Numero di schede RS232 installate
8	Non utilizzato
7,6	Numero di unità disco (00 → 1; 01 → 2; 10 → 3; 11 → 4 nel caso in cui bit 0 = 1)
5,4	Modalità video (00 non utilizzato; 01 = 40x25 scheda colore; 10 = 80x25 scheda colore; 11 = 80x25 Monocromatico)
3,2	RAM su piastra madre: (00 = 16Kb; 01 = 32Kb; 10 = 48Kb; 11 = 64Kb)
1	non utilizzato
0 = 1	nel caso in cui siano presenti unità disco esterne.

INT 12H – Determinazione della dimensione della memoria

Output
AX = numero di blocchi contigui di memoria da 1 Kb.

INT 13H Servizio 0 – Reinizializza sistema a dischi

Input
AH = 0

Output
Nessuno → AH = 0 Operazione corretta
Errore → AH = Codice errore (vedere Servizio 1)

***Nota**: Su sistemi dotati di disco fisso DL = 80H reinizializza le unità disco esterne; DL = 81H reinizializza il disco fisso.*

INT 13H Servizio 1 – Legge lo stato dell'ultima operazione eseguita

Input
AH = 1

Output (Codici di errore dei dischi)

AL = 00	Nessun errore
AL = 01	Trasferimento di comando errato
AL = 02	Marcatura indirizzo non trovata
AL = 03	Disco protetto da scrittura
AL = 04	Settore non trovato
AL = 05	Reset fallito
AL = 07	Parametri errati
AL = 09	DMA oltre il limite dei 64 Kb
AL = 0BH	Traccia difettosa
AL = 10H	CRC difettoso
AL = 11H	Errore di dati corretto
AL = 20H	Errore nel controller

AL = 40H	Errore nel posizionamento testina
AL = 80H	Nessuna risposta dal disco
AL = 0BBH	Errore non definito (generico)
AL = 0FFH	Operazione di sondaggio fallita

***Nota**: DL = Numero dell'unità disco; imposta il bit 7 a 1 per i dischi fissi. Per i dischi fissi il numero dell'unità disco di DL può essere compreso tra 80H e 87H.*

INT 13H Servizio 2 – Legge i settori del disco

Input
AH = 2
DL = Numero dell'unità disco
DH = Numero della testina
CH = Numero del cilindro o traccia dei dischi esterni
CL = I bit 7 e 6 nei due bit superiori di 10 bit del numero del cilindro
CL = Numero del settore (bit 0-5)
AL = Numero dei settori da leggere (Dischi esterni 1-8; disco fisso 1-80H; lettura/scrittura disco fisso esteso 1-79H)
ES:BX = Indirizzo dei buffer per la lettura/scrittura

Output
Nessuno → AL = Numero dei settori letti (disco esterno)
Errore → AH = Codice di errore su disco (vedere Servizio 1)

***Nota**: DL = Numero dell'unità disco; imposta il bit 7 a 1 per i dischi fissi, Per i dischi fissi il numero dell'unità disco indicato da DL è compreso tra 80H e 87H.*

INT 13H Servizio 3 – Scrive su settori del disco

Input
AH = 3
DL = Numero unità disco
DH = Numero testina
CH = Numero cilindro o traccia (dischi esterni)
CL = I bit 7 e 6 dei due bit superiori di 10 bit del numero del cilindro
CL = Numero del dettore (bit 0-5)
AL = Numero dei settori da scrivere (dischi esterni 1-8; Disco fisso 1-80H; lettura/scrittura su disco fisso esteso 1-79H)
ES:BX = Indirizzo del buffer di lettura/scrittura

Output
Nessuno → AL = Nessun setore scritto (su disco esterno)
Errore → AH = Codice di errore su disco (si veda il servizio 1)

***Nota**: DL = numero dell'unità disco; imposta il bit 7 a 1 per i dischi fissi. Per i dischi fissi il numero dell'unità disco è compreso tra 80H e 87H.*

INT 13 Servizio 4 – Verifica dei settori

Input
AH = 4
DL = Numero dell'unità disco
DH = Numero della testina
CH = Numero cilindro o traccia (dischi esterni)
CL = I bit 7 e 6 dei due bit superiori di 10 bit del numero del cilindro
CL = Numero del dettore (bit 0-5)
AL = Numero dei settori (dischi esterni 1-8; Disco fisso 1-80H; lettura/scrittura su disco fisso esteso 1-79H)

Output
Nessuno → AH = 0 Operazione correttamente eseguita
Errore → AH = Codice di errore su disco (si veda il servizio 1)

***Nota**: DL = numero dell'unità disco; imposta il bit 7 a 1 per i dischi fissi. Per i dischi fissi il numero dell'unità disco è compreso tra 80H e 87H.*

INT 13 Servizio 8 – Restituisce i parametri dell'unità disco attiva

Questo servizio è operativo solo sui dischi fissi e sui PS/2.

Input
AH = 8
DL = Numero dell'unità disco

Output
DL = Numero delle unità disco collegate al controller
DH = Valore massimo per il numero delle testine
CH = Valore massimo dei cilindri
CL = I bit 7 e 6 dei due bit superiori di 10 bit del numero del cilindro
CL = Valore massimo del numero dei settori (bit 0-5)
BL (solo per dischi esterni utilizzabili con un PS/2)
= 1 → unità disco da 360 Kb
= 2 → unità disco da 1,2 Mb
= 3 → unità disco da 720 Kb
= 4 → unità disco da 1,44 Mb

***Nota**: DL = Numero dell'unità disco; imposta il bit 7 a 1 per i dischi fissi. Per i dischi fissi il numero dell'unità disco viene compresa tra 80H e 87H.*

INT 13H Servizi 0AH e 0BH – Riservati

INT 13H Servizio 0CH – Spostamento testina (Seek) su un cilindro

Operativo SOLAMENTE su dischi fissi.

Input
AH = 0CH
DH = Numero testina
DL = Numero unità disco (intervallo valori ammesso: 80H-87H)
CH = Numero del cilindro
CL = Numero del settore; i bit 7 e 6 di CL corrispondono ai 2 bit superiori dee 10 bit del numero del cilindro

Output
Nessuno → AH = 0 Operazione terminata correttamente
Errore → AH = Codice errore su disco (si veda il Servizio 1)

***Nota**: DL = Numero dell'unità disco; imposta il bit 7 a 1 per i dischi fissi. Per i dischi fissi il numero delle unità disco in DL è compreso tra 80H e 87H.*

INT 13H Servizio 0DH – Reinizializzazione alternativa disco

INT 13H Servizio 0EH e 0FH – Riservato

INT 13H Servizio 10H – Controlla che il disco sia pronto

INT 13H Servizio 11H – Ritaratura del disco fisso

Questo servizio funziona SOLO con disco fisso.

Input
AH = 11H (Lettura)
DL = Numero unità disco (intervallo ammesso 80H-87H)

Output
Nessuno → AH = 0 Operazione eseguita correttamente
Errore → AH = Codice errore disco (si veda Servizio 1)

***Nota**: DL = Numero unità disco; imposta il bit 7 a 1 per i dischi fissi. Per i dischi fissi il numero dell'unità disco in DL può essere compreso nell'intervallo tra 80H e 87H.*

INT 13H – Servizi diagnostici

Questi servizi sono attivi SOLO su disco fisso.

Input
AH = 12H (Diagnostica RAM)
AH = 13H (Diagnostica unità disco)
AH = 14H (Diagnostica controller)
DL = Numero unità disco (intervallo ammesso: 80H-87H)

Output
Nessuno → AH = 0 Operazione eseguita correttamente
Errore → AH = Codice di errore disco (si veda Servizio 1)

***Nota**: DL = Numero unità disco; imposta il bit 7 a 1 per i dischi fissi. Per i dischi fissi il numero dell'unità disco in DL può essere compreso nell'intervallo tra 80H e 87H.*

INT 13H Servizio 19H – Parcheggia le testine (Solo PS/2)

Input (PS/2)
DL = Numero unità disco

Output
1 → Errore, AH = Codice errore
0 → Operazione eseguita correttamente

***Nota**: DL = Numero unità disco; imposta il bit 7 a 1 per i dischi fissi. Per i dischi fissi il numero dell'unità disco in DL può essere compreso nell'intervallo tra 80H e 87H.*

INT 14H, AH = 0 – Inizializza porta RS232

Input
AH = 0
Bit di AL:

0,1	Lunghezza parola (01 → 7 bit; 11 → 8 bit)
2	Bit di stop (0 → 1; 1 → 2 bit di stop)
3,4	Parità (00 → nessuna; 01 → dispari; 11 → pari)
5,6,7	Velocità baud:
	000 → 110
	001 → 150
	010 → 300
	011 → 600
	100 → 1200
	101 → 2400

110 → 4800
111 → 9600

INT 14H AH = 1 – Invia il carattere alla porta seriale

Input
AH = 1
AL = Carattere da inviare

Output
Se il bit 7 di AH è stato impostato, errore
Se il bit 7 non è stato impostato, i bit da 0 a 6 contengono lo stato (si veda INT 14H, AH = 3).

INT 14H AH = 2 – Riceve un carattere dalla porta seriale

Input
AH = 2
Output
AL = Carattere ricevuto
AH = 0 Operazione eseguita correttamente
In caso contrario AH contiene un codice di errore (si veda INT 14H, AH = 3)

INT 14H AH = 3 – Determina lo stato della porta seriale

Input
AH = 3

Output
Con i bit di AH impostati:

7	Errore di time-out (tempo scaduto)
6	Registro di scorrimento di trasmissione vuoto
5	Registro buffer di trasmissione vuoto
4	Riscontrato BREAK
3	Errore di inquadramento
2	Errore di parità
1	Errore di overrun
0	Dati pronti

Con i bit di AL impostati:

7	Riscontrato segnale linea di ricezione
6	Indicatore acustico
5	Data-Set-Ready (DSR)
4	Clear-To-Send (CTS)
3	Riscontrato segnale cambio linea ricezione
2	Riscontrato allarme di Training Edge
1	Delta Data-Set-Ready (DDSR)
0	Delta-Clear-To-Send (DCTS)

INT 15H – I/O cassetta

Input

AH = 0 Avvia il motore della cassetta
AH = 1 Disattiva il motore della cassetta
AH = 2 Legge uno o più blocchi da 256 byte. memorizza i dati in ES:BX. CX = Conteggio dei byte da leggere
AH = 3 Scrive uno o più blocchi da 256 byte da ES:BX. Conteggio dei byte da scrivere in CX.

Output

DX = Numero dei byte effettivamente letti
Imposta il flag di riporto in caso di errore.
AH = 01 Errore CRC
AH = 02 Trasferimento dati perduto
AH = 04 Nessun dato trovato

***Nota**: Nelle recenti versioni del BIOS sono state aggiunte nuove voci a questo interrupt come, per esempio, la gestione del joystick, la possibilità di passare alla modalità processore (protetta o meno), la gestione del mouse e alcuni parametri del BIOS.*

INT 16H Servizio 0 – Legge un tasto da tastiera

Input

AH = 0

Output

AH = Scan Code
AL = Codice ASCII

INT 16H Servizio 1 – Determina se un carattere è pronto per essere letto

Input

AH = 1

Output

Flag Zero = 1 Buffer vuoto
Flag Zero = 0 AH = Scan code
AL = Codice ASCII

INT 16H Servizio 2 – Determina lo stato della tastiera

Input

AH = 2

Output
AL = Byte di stato della tastiera

INT 17H Servizio 0 – Stampa carattere in AL

Input
AH = 0
AL = Carattere da stampare
DX = Numero della stampante (0, 1, 2)

Output
AH = 1 Time-out stampante (tempo scaduto)

INT 17H Servizio 1 – Inizializzazione porta stampante

Input
AH = 1
DX = Numero della stampante (0, 1, 2)

Output
AH = Stato della stampante
Bit impostati di AH:

7	Stampante non occupata
6	Accettazione
5	Mancanza carta
4	Stampante selezionata
3	Errore di I/O
2	Non utilizzato
1	Non utilizzato
0	Errore di time-out (tempo scaduto)

INT 17H Servizio 2 – Legge lo stato della stampante in AH

Input
AH = 2
DX = Numero della stampante (0, 1, 2)

Output
AH impostato al byte di stato come per INT 17H, AH = 1

INT 18H – BASIC residente

Questo interrupt avvia il basic residente nella ROM del PC.

INT 19H – Reinizializza il computer (Bootstrap)

Questo interrupt avvia il computer (si provi con DEBUG)

INT 1AH Servizio 0 – Legge l'ora corrente

Input
AH = 0

Output
CX = Conteggio impulsi della parola alta
DX = Conteggio impulsi della parola bassa
AL = 0 nel caso in cui il timer non abbia completato un ciclo di 24 ore dopo l'ultima lettura

***Nota**: Il conteggio degli impulsi aumenta di 65.536 unità in un'ora.*

INT 1AH Servizio 1 – Imposta ora corrente

Input
AH = 1
CX = Conteggio impulsi della parola alta
DX = Conteggio impulsi della parola bassa

***Nota**: Il conteggio degli impulsi aumenta di 65.536 unità in un'ora.*

INT 1BH – Indirizzo di BREAK da tastiera

INT 1CH – Interruzione impulsi del timer

INT 1DH – Tabella dei parametri video

INT 1EH – Parametri del disco esterno

INT 1FH – Definizione caratteri grafici

INTERRUPT DEL DOS

L'interrupt 1FH è l'ultimo interrupt del BIOS e quelli del DOS iniziano con INT 20H.

INT 20H – Termina

I programmi generalmente vengono conclusi da INT 20H.

INT 21H

L'interrupt 21H è l'interrupt di servizio del DOS. Per richiamare uno di questi servizi caricare AH con il numero del servizio desiderato e verranno visualizzati gli altri registri disponibili.

INT 21H Servizio 0 – Termina programma

Input
AH = 0

INT 21H Servizio 1 – Input da tastiera

AH = 1

Output
AL = Codice ASCII del tasto premuto che viene visualizzato sul video

***Nota**: Verifica la presenza di ^C oppure di ^BREAK.*

INT 21H Servizio 2 – Output a video di carattere

Input
AH = 2
DL = Codice carattere ASCII

INT 21H Servizio 3 – Input standard di periferica ausiliaria

Input
AH = 3

Output
Carattere in AL

INT 21H Servizio 4 – Output standard a periferica ausiliaria

Input
AH = 4
DL = Carattere da emettere

INT 21H Servizio 5 – Output stampante

Input
AH = 5
DL = Carattere da emettere

INT 21H Servizio 6 – I/O Console senza echo

Input	**Output**
AH = 6	
DL = FFH	Flag zero impostato se nessun carattere era pronto. Altrimenti AL contiene il codice ASCII del carattere
DL < FFH	Emette a video il codice ASCII contenuto in DL

***Nota**: Non verifica la presenza di ^C oppure di ^BREAK.*

INT 21H Servizio 7 – Input diretto da console senza echo

Input
AH = 7

Output
AL = Codice ASCII del tasto premuto senza visualizzazione a video

***Nota**: Non verifica la presenza di ^C oppure di ^BREAK.*

INT 21H Servizio 8 – Input da console senza echo con controllo di ^C

Input
AH = 8

Output
AL = Codice ASCII del tasto premuto senza visualizzazione del carattere

***Nota**: Verifica la presenza di ^C o di ^BREAK.*

INT 21H Servizio 9 – Stampa di una stringa

Input
DS:DX punta a una stringa che termina con "$".
AH = 9

INT 21H Servizio A – Input stringa

Input
AH = 0AH
[DS:DX] = Ampiezza del buffer

Output
Buffer di DS:DX completo
Echo dei tasti premuti

***Nota**: Controlla la presenza di ^C o di ^BREAK.*

INT 21H Servizio 0BH – Controlla lo stato dell'input

Input
AH = 0BH

Output
AL = FF Carattere pronto
AL = 00 Nulla da leggere

***Nota**: Viene verificata la presenza di ^BREAK.*

INT 21H Servizio 0CH – Svuota il buffer di tastiera e richiama il servizio

Input
AH = 0CH
AL = N. funzione di tastiera

Output
Output standard per il servizio selezionato

***Nota**: Viene verificata la presenza di ^BREAK.*

INT 21H Servizio 0DH – Reinizializzazione disco

Input
AH = 0DH

INT 21H Servizio 0EH – Selezione disco

Input
AH = 0EH
DL = N. unità disco (DL = 0: A; DL = 1: B; e così via)

INT 21H Servizio 0FH – Apertura di un file preesistente

Input
DS:DX punta a un FCB
AH = 0FH

Output
AL = 0 Operazione eseguita correttamente
AL = FF Errore

INT 21H Servizio 10H – Chiude file

Input
DS:DX punta a un FCB
AH = 10H

Output
AL = 0 Operazione eseguita correttamente
AL = FF Errore

INT 21H Servizio 11H – Ricerca il primo file corrispondente

Input
DS:DX Punta a un FCB non aperto
AH = 11H

Output
AL = FF Errore
AL = 0 Operazione eseguita correttamente. Il DTA contiene l'FCB corrispondente

***Nota**: Notare che DTA è a CS:0080 nei file .COM al momento dell'avviamento degli stessi.*

INT 21H Servizio 12H – Ricerca il file corrispondente successivo

Input
DS:DX punta a un FCB non aperto
AH = 12H

Output
AL = FF Errore
AL = 0 Operazione eseguita correttamente. DTA contiene l'FCB corrispondente

***Nota**: Utilizzare questo servizio dopo il servizio 11H.*

INT 21H Servizio 13H – Cancella file

Input
DS:DX punta a un FCB non aperto
AH = 13H

Output
AL = FF Errore
AL = 0 Operazione eseguita correttamente

INT 21H Servizio 14H – Lettura sequenziale

Input
DS:DX punta a un FCB aperto
AH = 14H
Blocco attivo e record impostati in FCB

Output
Vengono richiesti i record in DTA
AL = 0 Operazione eseguita correttamente
AL = 1 Fine file, nessun dato nel record
AL = 2 Segmento DTA troppo piccolo per il record
AL = 3 Fine file, record completato con una serie di byte zero

***Nota**: L'indirizzo del record viene aumentato.*

INT 21H Servizio 15H – Scrittura sequenziale

Input
DS:DX punta a un FCB aperto
AH = 15H
Blocco corrente e record impostati in FCB

Output
Lettura di un record dal DTA e sua scrittura
AL = 0 Operazione eseguita correttamente
AL = 1 Disco pieno
AL = 2 Segmento DTA troppo piccolo per il record

***Nota**: L'indirizzo del record viene aumentato.*

INT 21H Servizio 16H – Crea file

Input
DS:DX punta a un FCB non aperto
AH = 16H

Output
AL = 0 Operazione eseguita correttamente
AL = FF Directory piena

INT 21H Servizio 17H – Rinomina file

Input
DS:DX punta a un FCB modificato
AH = 17H

Output
AL = 0 Operazione eseguita correttamente
AL = FF Errore

***Nota**: Negli FCB modificati il nome del secondo file inizia 6 byte dopo la fine del nome del primo in DS:DX + 11H.*

INT 21H Servizio 18H – Interno al DOS

INT 21H Servizio 19H – Riporta il disco attivo

Input
AH = 19H

Output
AL = Disco attivo (0 = A; 1 = B; ecc.)

INT 21H Servizio 1AH – Imposta l'area di trasferimento del disco (DTA)

Input
DS:DX punta a un nuovo indirizzo DTA
AH = 1AH

***Nota**: DTA* (Disk Trasfer Address, *Area di trasferimento del disco*) *è l'area utilizzata con i servizi FCB. Il DTA di default ha un'estensione di 128 byte e inizia a CS:0080 nel PSP.*

INT 21H Servizio 1BH – Informazioni della FAT per l'unità disco di default

Input
AH = 1BH

Output
DS:BX punta al byte della FAT
DX = Numero dei cluster
AL = Numero dei settori/cluster
CX = Dimensione di un settore (512 byte)

***Nota**: I file vengono memorizzati nei cluster, unità minima di allocazione su disco.*

INT 21H Servizio 1CH – Informazioni FAT per l'unità disco specificata

Input
AH = 1CH
DL = Numero di unità (0 = defualt; 1 = A, ecc.)

Output
DS:BX punta al byte della FAT
DX = Numero dei cluster
AL = Numero di settori/cluster
CX = Dimensione di un settore (512 byte)

***Nota**: I file vengono memorizzati nei cluster, unità minima di allocazione su disco.*

INT 21H Servizi 1DH-20H – Interni al DOS

INT 21H Servizio 21H – Lettura casuale

Input
DS:DX punta a un FCB aperto

Imposta il campo di un record casuale dell'FCB in DS:DX + 33 e DS:DX + 35
AH = 21H

Output
AL = 00 Operazione eseguita correttamente
AL = 01 Fine file, nessun dato ulteriore
AL = 02 Spazio non sufficiente nel segmento DTA
AL = 03 Fine file, record parziale completato da una serie di zero

INT 21H Servizio 22H – Scrittura casuale

Input
DS:DX punta a un FCB aperto
Imposta il campo di un record casuale dell'FCB in DS:DX + 33 e DS:DX + 35
AH = 21H

Output
AL = 00 Operazione eseguita correttamente
AL = 01 Disco pieno
AL = 02 Spazio non sufficiente nel segmento DTA

INT 21H Servizio 23H – Dimensione file

Input
DS:DX punta a un FCB non aperto
AH = 23H
Output
AL = 00 Operazione eseguita correttamente
AL = FF Nessun file trovato corrispondente a FCB. Viene arrotondato il campo del record casuale impostato alla lunghezza del file nei record

INT 21H Servizio 24H – Imposta il campo del record casuale

Input
DS:DX punta a un FCB aperto
AH = 24H

Output
Campo del record casuale impostato in modo che corrisponda la record e al blocco attivo

INT 21H Servizio 25H – Imposta il vettore dell'interrupt

Input
AH = 25H
AL = Numero di interrupt
DS:DX = Nuovo indirizzo

Nota: Questo servizio può aiutare l'utente a intercettare il vettore dell'interrupt.

INT 21H Servizio 26H – Crea un nuovo segmento di programma (PSP)

INT 21H Servizio 27H – Lettura di blocco casuale

Input
DS:DX punta a un FCB aperto
Imposta il campo di un record casuale dell'FCB in DS:DX + 33 e DS:DX + 35
AH = 27H

Output
AL = 00 Operazione eseguita correttamente
AL = 01 Fine file, nessun dato ulteriore
AL = 02 Spazio non sufficiente nel segmento in DTA
AL = 03 Fine file, il record parziale viene completato con una serie di zero

CX = Numero dei record letti. I campi del record casuale vengono impostati per poter accedere al record successivo

Nota: Il buffer dei dati utilizzato nei servizi FCB è la DTA (area di trasferimento del disco).

INT 21H Servizio 28H – Scrittura su blocco casuale

Input
DS:DX punta a un FCB aperto
Imposta il campo di record casuale dell'FCB in DS:DX + 33 e DS:DX + 35
CX = Numero di record da scrivere
AH = 28H

Output
AL = 00 Operazione eseguita correttamente
AL = 01 Disco pieno
AL = 02 Spazio non sufficiente nel segmento DTA
I campi del record casuale vengono impostati per poter accedere al record successivo.

Nota: CX = 0 corrisponde al file da impostare alle dimensioni indicate dal campo di record casuale. Il buffer di dati utilizzato nei servizi FCB è la DTA (area di trasferimento disco).

INT 21H Servizio 29H – Analisi del nome del file

Input
DS:SI = Riga di comando da analizzare
ES:DI = Indirizzo al quale inviare l'FCB
AL = Bit 0=1 Viene effettuata l'analisi dei separatori di riempimento
Bit 1=1 L'ID dell'unità disco nell'FCB finale viene modificata SOLO se è stata specificata un'unità disco
Bit 2=1 Il nome del file in FCB viene modificato SOLO se la riga di comando include il nome del file
Bt 3=1 L'estensione del nome del file in FCB viene modificata SOLO se la riga di comando contiene l'estensione di un nome di file
AH = 29H

Output
DS:SI = Primo carattere dopo il nome del file
ES:DI = FCB valido

***Nota**: Se la riga di comando non contiene un corretto nome di file, ES:[DI+1] risulterà vuoto.*

INT 21H Servizio 2AH – Acquisisce la data

Input
AH = 2AH

Output
CX = Anno - 1980
DH = Mese (1 = gennaio, ecc.)
DL = Giorno del mese

INT 21H Servizio 2BH – Imposta la data

Input
CX = Anno - 1980
DH = Mese (1 = gennaio, ecc.)
DL = Giorno del mese
AH = 2BH

Output
AL = 0 Operazione eseguita correttamente
AL = FF Data non valida

INT 21H Servizio 2CH – Acquisisce l'ora

Input
AH = 2CH

Output
CH = Ore (0-23)
CL = Minuti (0-59)
DH = Secondi (0-59)
DL = Centesimi di secondo (0-99)

INT 21H Servizio 2DH – Imposta l'ora

Input
AH = 2DH
CH = Ore (0-23)
CL = Minuti (0-59)
DH = Secondi (0-59)
DL = Centesimi di secondo (0-99)

Output
AL = 0 Operazione eseguita correttamente
AL = FF Ora non valida

INT 21H Servizio 2EH – Imposta il flag di verifica

Input
AH = 2EH
DL = 0
AL = 1 Attiva la verifica
AL = 0 Disattiva la verifica

INT 21H Servizio 2FH – Acquisisce DTA attivo

Input
AH = 2FH

Output
ES:BX = Indirizzo DTA attivo.

***Nota**: Il buffer dei dati dei servizi FCB è il DTA (Area di trasferimento disco).*

INT 21H Servizio 30H – Acquisisce numero di versione DOS

Input
AH = 30H

Output
AL = Numero superiore della versione (per esempio 3 in 3.10)
AH = Numero inferiore della versione (per esempio 10 in 3.10)
BX = 0
CX = 0

***Nota**: Se AL restituisce 0, questo significa che si sta lavorando con una versione del DOS precedente alla 2.0.*

INT 21H Servizio 31H – Termina il processo e rimane residente

Input
AH = 31H
AL = Codice binario di uscita
DX = Dimensione della memoria richiesta per i paragrafi

***Nota**: Il codice di uscita può essere letto da un programma di livello superiore con il servizio 4DH. Può inoltre essere controllato dai comandi ERRORLEVEL presenti nei file batch.*

INT 21H Servizio 32H – Interno al DOS

INT 21H Servizio 33H – Controlla presenza ^BREAK

Input
AH = 33H
AL = 0 Verifica lo stato del controllo di ^BREAK
AL = 1 Imposta lo stato del controllo di ^BREAK
(DL = 0 lo disattiva - DL = 1 lo attiva)

Output
DL = 0 Off
DL = 1 On

INT 21H Servizio 34H – Interno al DOS

INT 21H Servizio 35H – Acquisisce vettore interrupt

Input
AH = 35H
AL = Numero di interrupt

Output
ES:BX = Vettore di interrupt

INT 21H Servizio 36H – Acquisisce spazio libero su disco

Input
AH = 36H
DL = Numero di unità (0 = Default; 1 = A, ecc.)

Output
AX = 0FFFH Numero di unità non valido
AX = Numero di settori/cluster

BX = Numero di cluster disponibili
CX = Dimensione di un settore (512)
DX = Numero di cluster

***Nota**: I file vengono memorizzati in cluster (unità minima di allocazione su disco).*

INT 21H Servizio 37H – Interno al DOS

INT 21H Servizio 38H – Restituisce informazioni dipendenti dal paese

Input
AH = 38H
DS:DX = Indirizzo di un blocco da 32 byte
AL = 0

Il blocco da 32 byte appare nel modo seguente:
2 byte formato DATA/ORA
1 byte simbolo valuta (ASCII)
1 byte impostato a 0
1 byte separatore di migliaia (ASCII)
1 byte impostato a 0
1 byte separatore decimale (ASCII)
1 byte impostato a 0
14 byte utilizzati all'interno

Il formato DATA/ORA assume i seguenti valori:
0 = USA (H:M:S M/D/Y)
1 = Europa (H:M:S D/M/Y)
2 = Giappone (H:M:S D:M:Y)

Output
Riempimento del blocco da 32 byte (vedi sotto).

***Nota**: Nel DOS 3+ è possibile impostare questi valori come essi vengono letti.*

INT 21H Servizio 39H – Crea una sottodirectory

Input
AH = 39H
DS:DX punta alla stringa ASCIIZ che forma il nome della directory

Output
Nessuno: Operazione eseguita correttamente
Errore: AH contiene il valore di errore

AH = 3 percorso non trovato
AH = 5 accesso negato

INT 21H Servizio 3AH – Cancella una sottodirectory

Input
AH = 3AH
DS:DX punta alla stringa ASCIIZ che forma il nome della directory

Output
Nessuno: operazione eseguita correttamente
Errore: AH contiene il valore di errore
AH = 3 percorso non trovato
AH = 5 accesso negato o sottodirectory non vuota

INT 21H Servizio 3BH – Cambia la directory attiva

Input
AH = 3BH
DS:DX punta alla stringa ASCIIZ che forma il nome della directory

Output
Nessuno: Operazione eseguita correttamente
Errore: AH contiene il valore di errore
AH = 3 percorso non trovato

INT 21H Servizio 3CH – Crea un file

Input
DS:DX punta al nome del file ASCIIZ
CX = Attributo del file
AH = 3CH

Output
Nessuno: AH = Identificatore del file
Errore: AL = 3 percorso non trovato
AL = 4 troppi file aperti
AL = 5 directory piena o esiste un file precedente di sola lettura

INT 21H Servizio 3DH – Apre un file

Input
DS:DX = Punta a un nome di file ASCIIZ
AL = codice di accesso
AH = 3DH

Codici di accesso
AL = 0 File aperto per sola lettura

AL = 1 File aperto per operazioni di scrittura
AL = 2 File aperto per operazioni di lettura/scrittura

Codice di accesso DOS 3+: isssraaa
i = 1 Il file non deve essere ereditato da un processo di livello inferiore
i = 0 L'identificatore del file verrà ereditato
sss=000 Modo di condivisione
sss=001 Impedisce qualsiasi accesso
sss=010 Impedisce l'accesso per scrittura
sss=011 Impedisce l'accesso per la lettura
sss=100 Non impedisce le operazioni di lettura/scrittura
r = Riservato
aaa=000 Accesso di sola lettura
aaa=001 Accesso di sola scrittura
aaa=010 Accesso per operazioni di lettura/scrittura

Output
Nessuno: AX = Identificatore del file
Errore: AL = Codice di errore (controllare tabella errori)

INT 21H Servizio 3EH – Rilascia l'identificatore di un file

Input
BX contiene l'identificatore del file valido
AH = 3EH

Output
Errore: AL = 6 identificatore non valido

INT 21H Servizio 3FH – Legge da file o da periferica

Input
DS:DX = Indirizzo del buffer di dati
CX = Numero dei byte da leggere
BX = Identificatore del file
AH = 3FH

Output
Nessuno: AX = Numero dei byte letti
Errore: AL = 5 Accesso negato
AL = 6 Identificatore non valido

INT 21H Servizio 40H – Scrive su file o periferica

Input
DS:DX Indirizzo del buffer di dati
CX = Numero di byte da scrivere

BX = Identificatore del file
AH = 40H

Output
Nessuno: AH = Numero dei byte scritti
Errore: AL = 5 Accesso negato
AL = 6 Identificatore non valido

***Nota**: Un disco pieno non viene considerato come errore: si controlli il numero dei byte che si desidera scrivere (CX) in rapporto al numero di byte già scritti (riportati in AX). Se questi non corrispondono, il disco, probabilmente, è pieno.*

INT 21H Servizio 41H – Cancella un file

Input
DS:DX = Nome del file (ASCIIZ)
AH = 41H

Output
Nessuno: Operazione eseguita correttamente
Errore: AL = 2 File non trovato
AL = 5 Accesso negato

***Nota**: Non sono ammessi i caratteri jolly all'interno del nome del file.*

INT 21H Servizio 42H – Sposta il puntatore di lettura/scrittura

Input
BX = Identificatore del file
CX:DX = Offset richiesto
AL = Posizione di riferimento
AH = 42H

Posizione di riferimento
AL = 0 Il puntatore di lettura/scrittura viene spostato di CX:DX dall'inizio del file
AL = 1 Il puntatore viene aumentato di CX:DX byte
AL = 2 Identificatore non valido

Output
Nessuno: DX:AX Nuova posizione del puntatore
Errore: AL = 1 Numero di funzione non ammesso
AL = 6 Identificatore non valido

INT 21H Servizio 43H – Modifica l'attributo di un file

Input
DS:DX = Stringa del nome del file (ASCIIZ)
AL = 1 Attributo del file modificato: CX contiene il nuovo attributo
AL = 0 Attributo del file restituito da CX
AH = 43H

Output
Nessuno: Operazione eseguita correttamente
Errore: AL = 2 File non trovato
AL = 3 Percorso non trovato
AL = 5 Accesso negato
Se AL = 0 CX restituirebbe l'attributo.

INT 21H Servizio 44H – Controllo I/O

INT 21H Servizio 45H – Duplica l'identificatore di file

Input
BX = Identificatore del file da duplicare
AH = 45H

Output
Nessuno: AX = Nuovo identificatore duplicato
Errore: AL = 4 Troppi file aperti
AL = 6 Identificatore non valido

INT 21H Servizio 46H – Forza la duplicazione dell'identificatore di un file

Input
BX = Identificatore del file da duplicare
CX = Secondo identificatore del file
AH = 46H

Output
Nessuno: Gli identificatori fanno riferimento allo stesso "flusso"
Errore: AL = 6 Identificatore non valido

INT 21H Servizio 47H – Acquisisce la directory attiva su un'unità specificata

Input
AH = 47H
DS:SI punta a un buffer di 64 byte
DL = numero dell'unità disco

Output
Nessuno: Operazione eseguita correttamente (ASCIIZ a DS:SI)
Errore: AH = 15 Unità disco specificata non valida

***Nota**: La lettera dell'unità disco non comprende la stringa riportata in ASCIIZ.*

INT 21H Servizio 48H – Allocazione di memoria

Input
AH = 48H
BX = Numero di paragrafi

Output
Nessuno: AX:0000 indirizzo del blocco di memoria
Errore: AL = 7 Blocchi di controllo della memoria danneggiati
AL = 8 Memoria insufficiente. BX contiene la quantità massima richiesta disponibile

INT 21H Servizio 49H – Memoria allocata libera

Input
AH = 49H
ES = Segmento di blocco in corso di liberazione

Output
Nessuno: Operazione eseguita correttamente
Errore: AL = 7 Blocchi di controllo di memoria danneggiati
AL = 9 Indirizzo del blocco di memoria errato

INT 21H Servizio 4AH – Imposta blocco

Input
AH = 4AH
ES = Segmento del blocco da modificare
BX = Dimensione richiesta in paragrafi

Output
Nessuno: Operazione eseguita correttamente
Errore: AL = 7 Blocchi di controllo della memoria danneggiati
AL = 8 Memoria indufficiente. BX contiene la richiesta massima possibile
AL = 9 Indirizzo del blocco di memoria errato

INT 21H Servizio 4BH – Carica o esegue un programma (EXEC)

Input
AH = 4BH
DS:DX = Stringa ASCIIZ con unità disco, percorso e nome file

ES:BX = Indirizzo del parametro del blocco (vedi sotto)
AL = 0 Carica ed esegue il programma
AL = 3 Carica ma non crea PSP, non avvia (Overlay) (Vedi sotto)

Parametro blocco per AL = 0
Indirizzo segmento stringa ambiente tra trasferire (Word)
Indirizzo del comando da inserire in PSP+80H (DWord)
Indirizzo di default FCB da inserire in PSP+5CH (DWord)
Indirizzo del secondo default FCB da inserire in PSP+6CH (DWord)

Parametro blocco per AL = 3
Indirizzo segmento al quale caricare il file (Word)
Fattore di riallocazione dell'immagine (Word)

Output
Nessuno: Operazione eseguita correttamente
Errore: AL = 1 Numero di funzione non valido
AL = 2 File non trovato su disco
AL = 5 Accesso negato
AL = 8 Memoria insufficiente per le operazioni richieste
AL = 10 Ambiente non valido
AL = 11 Formato non valido

INT 21H Servizio 4CH – Uscita

Input
AH = 4CH
AL = Codice binario di ritorno

***Nota**: Questo servizio può concludere un programma.*

INT 21H Servizio 4DH – Acquisisce codice di ritorno di un sotto-processo

Input
AH = 4DH

Output
AL = Codice di ritorno binario del sottoprocesso
AH = 0 se il sottoprocesso termina normalmente
AH = 1 se il sottoprocesso termina con ^BREAK
AH = 2 se il sottoprocesso termina con un errore critico di periferica
AH = 3 se il sottoprocesso termina con un servizio 31H

INT 21H Servizio 4EH – Trova il primo file che corrisponde

Input
DS:DX = Stringa del file ASCIIZ
CX = Attributo che deve corrispondere
AH = 4EH

Output
Errore: AL = 2 Nessuna corrispondenza trovata
AL = 18 Nessun ulteriore file reperito
Nessuno: DTA completata come segue:
21 byte riservati
1 byte attributo reperito
2 byte ora del file
2 byte data del file
2 byte dimensione parola inferiore
2 byte dimesnione parola superiore
13 byte nome ed estensione del file trovato in forma ASCIIZ (senza nome percorso)

***Nota**: Il buffer dei dati utilizzato nei servizi FCB è il DTA (area trasferimento disco). Vedi servizi precedenti.*

INT 21H Servizio 4FH – Trova il file corrispondente successivo

Input
Utilizza servizio 4EH prima di 4FH.
AH = 4FH

Output
Errore: AL = 18 nessun altro file
Nessuno: DTA completata come segue:
21 byte riservati
1 byte attributo reperito
2 byte ora del file
2 byte data del file
2 byte dimensione parola inferiore
2 byte dimensione parola superiore
13 byte nome ed estensione del file trovato in forma ASCIIZ (senza nome percorso)

***Nota**: Il buffer dei dati utilizzato nei servizi FCB è il DTA (area di trasferimento disco). Vedi servizi precedenti.*

INT 21H Servizio 50H-53H – Interni al DOS

INT 21H Servizio 54H – Acquisisce stato di verifica

Input
AH = 54H

Output
AL = 0 Verifica disattivata
AL = 1 Verifica attivata

INT 21H Servizio 55H – Interno al DOS

INT 21H Servizio 56H – Rinomina file

Input
DS:DX = Stringa del file ASCIIZ a cui deve essere assegnato un nuovo nome
ES:DI = stringa del file ASCIIZ contenente il nuovo nome
AH = 56H

Output
Nessuno: Operazione eseguita correttamente
Errore: AL = 3 Percorso non trovato
AL = 5 Accesso negato
AL = 17 Periferica diversa

***Nota**: Al file NON PUÒ essere assegnato il nome di una diversa unità disco.*

INT 21H Servizio 57H – Acquisisce e imposta la data e l'ora di un file

Input
BX = gestore del file
AL = 0 Acquisisce data e ora
AL = 1 Imposta l'ora in CX e la data in DX

Output
Nessuno: CX restituisce l'ora; DX restituisce la data

Data e ora del file impostate.

Errore: AL = 1 Numero di funzione non valida
AL = 6 Gestore non valido

La data e l'ora del file vengono memorizzate come segue:
Ora = 2048 x ore + 32 x minuti + secondi/2
Data = 512 x (anno - 1980) + 32 x mese + giorno

INT 21H Servizio 58H – Interno al DOS

INT 21H Servizio 59H – Acquisisce errore esteso DOS 3+

Input
AH = 59H
BX = 0

Output
AX = Errore esteso
BX = Classe di errore
BL = Azione suggerita
CH = Codice di luogo

***Nota**: Questo servizio per la gestione dell'errore e molto lungo e coinvolge la maggior parte degli errori estesi del DOS 3+.*

INT 21H Servizio 5AH – Crea un solo file DOS 3+

Input
AH = 5AH
DS:DX = Indirizzo di un percorso ASCIIZ chiuso da "\"
CX = Attributo del file

Output
AX = Errore se il flag di riporto è impostato
DS:DX = Percorso e nome del file ASCIIZ

INT 21H Servizio 5BH – Crea un nuovo file DOS3+

Input
AH = 5BH
DS:DX = Indirizzo di un percorso ASCIIZ chiuso da "\"
CX = Attributo del file

Output
AX = Errore se è stato impostato il flag di riporto
AX = Gestore se il flag di riporto non è stato impostato

INT 21H Servizio 5CH – Chiude e apre l'accesso a un file DOS 3+

Input
AH = 5CH
AL = 0 Blocca l'intervallo di byte
AL = 1 Sblocca l'intervallo di byte
BX = Gestore del file
CX = Avvia l'intervallo dei byte (parola superiore)
DX = Avvia l'intervallo dei byte (parola inferiore)

SI = Nessun bite da bloccare o sbloccare (parola superiore)
DI = Nessun bite da bloccare o sbloccare (parola inferiore)

Output
Se riporto = 1, AX = Errore

INT 21H Servizio 5E00H – Acquisisce il nome della macchina DOS 3+

Input
AX = 5E00H
DS:DX = Buffer per nome macchina

Output
DS:DX = Nome macchina ASCIIZ
CH = 0 Nome non definito
CL = Numero NETBIOS
AX = Errore se impostato il flag di riporto

INT 21H Servizio 5E02 – Imposta setup della stampante DOS 3+

Input
AX = 5E02H
BX = Ridirezione l'indice dell'elenco
CX = Lunghezza della stringa di setup
DS:DI = Puntatore al buffer del setup stampante

Output
AX = Errore se il flag di riporto è impostato

INT 21H Servizio 5E03 – Acquisisce setup di stampante DOS 3+

Input
AX = 5E03H
BX = Ridireziona l'indice dell'elenco
ES:DI = Puntatore al buffer del setup stampante

Output
AX = Errore se il flag di riporto è impostato
CX = Lunghezza dei dati restituiti
ES:DI = Riempito con la stringa di setup della stampante

INT 21H Servizio 5F03 – Redirezione periferica DOS 3+

Input
AX = 5F03H
BL = Tipo di periferica
BL = 3 Periferica stampante

BL = 4 Periferica file
CX = Valore da salvare per chiamante
DS:SI = Sorgente ASCIIZ nome della periferica
ES:DI = Destinazione del percorso di rete ASCIIZ con password

Output
AX = Errore se il flag di riporto è impostato

INT 21H Servizio 5F04H – Cancella redirezionamento DOS 3+

Input
AX = 5F04H
DS:SI = None o percorso della periferica ASCIIZ

Output
AX = Errore se è stato impostato il flag di riporto

INT 21H Servizio 62H – Acquisisce il prefisso segmento di programma DOS 3+

Input
AX = 62H

Output
BX = Segmento del programma esecutivo attivo

INT 21H Servizio 67H – Imposta il conteggio degli handle 3.30

Input
AX = 67H
BX = Numero consentito di gestori aperti (fino a 255)

Output
AX = Erriore se il flag di riporto è impostato

INT 21H Servizio 68H – Impegna un file (scrive nel buffer) DOS 3.30

Input
AX = 68H

Output
BX = Gestore del file

***Nota**: 68H è l'ultimo dei servizi dell'interrupt 2iH del DOS 3.30.*

INT 22H – Conclude indirizzo

INT 23H – Indirizzo dell'uscita ^BREAK

INT 24H – Gestore dell'errore critico

AH viene riempito come segue:
0 Disco protetto
1 Unità sconosciuta
2 L'unità disco richiesta non è pronta
3 Comando sconosciuto
4 Ridondanza ciclica di controllo degli errori nei dati
5 Richiesta scorretta di lunghezza struttura
6 Sicerca errore
7 Tipo supporto sconosciuto
8 Settore non trovato
9 Stampante senza carta
A Errore di scrittura
B Errore di lettura
C Errore generale

Se viene semplicemente eseguito IRET, il DOS avvierà un'operazione basata unicamente sul contenuto di AL. Se AL = 0 l'errore verrà ignorato. Se AL = 1 l'operazione verrà ripetuta. Se AL = 2 il programma verrà concluso tramite l'invocazione di INT 23H.

INT 25H – Lettura assoluta del disco

Input
AL = Numero unità disco
CX = Numero settori da leggere
DX = Primo settore logico
DS:BX = Indirizzo del buffer

Output
Nessuno: Operazione eseguita correttamente

Errore:	AH = 80H	Il disco non risponde
	AH = 40H	Ricerca fallita
	AH = 20H	Errore del controller
	AH = 10H	Controllo dell'errore di CRC scorretta
	AH = 08	Overrun DMA
	AH = 04	Settore non trovato
	AH = 03	Errore di protezione da scrittura
	AH = 02	Manca segnalazione di indirizzo
	AH = 00	Errore sconosciuto

INT 28H-2EH – Interni al DOS

INT 2FH – Interrupt multiplex

INT 30H-3FH – Riservati al DOS

INT 40H-5FH – Riservati

INT 60H-67H – Riservati per utenti software

INT 68H-7FH – Non utilizzati

INT 80H-85H – Riservati dal BASIC

INT 86H-F0H – Utilizzati dall'interprete del BASIC

INT F1H-FFH – Non utilizzato

INDICE ANALITICO

!

.CODE, direttiva per l'assembler 436
.COM, file 435
.DATA, direttiva per l'assembler 441

A

Add Wztch, opzione 367
ADD, istruzione 448
albero binario 327
Array 316
  Dinamico 246
Assemble, comando 426
assembler 423
assembler, direttive 436
assembly, linguaggio 421-473
Attr% 40, 47
Attributo video 47

B

BarAttrb% 134
Barra dei menu 127
  Attivazione 141
  Rimuovere la 149
  Visualizzazione 141
BASIC
  PDS 84
  Professional Development System XII
  Versione 7.0 XII
BC.EXE 85, 200
BIOS 387
  modo schermo 389
BP (Base Pointer) 480
breakpoint 367, 376
buffer circolare 326
Byte 11

C

cicli
  in linguaggio assembly 468
cluster 320
CMP, istruzione 450
coda
  in un buffer 326
CodeView 374
Codice ASCII esteso 5
commenti
  in linguaggio assembly 446
Controllo scheda grafica installata 266
convenzione di chiamata BASIC 478

MouseInformation( ) 99, 103
MouseMoveCursor( ) 104
MouseShowCursor( ) 93, 95
MOUSESYS.COM 91
MouseTimesPressed( ) 107
MouseTimesReleased( ) 110
MouseVerticalRange( ) 116
MOV, istruzione 423
MOVE, istruzione 290
MUL, istruzione 461

N

NCol% 40, 47
notazione in complemento
a due 482
NRow% 40, 47

O

Offset 223
OFFSET, direttiva per l'assembler 445
ordine di inserimento 276

P

paragrafo 432
passaggio dei parametri 478
POP, istruzione 463
PROC, direttiva per l'assembler 469
procedure
in linguaggio assembly 469
Programma
Frammentazione 29
Programmazione modulare 41, 127
Programmazione sequenziale 215
Programmazione top-down 215
PSET 226
puntatori
nelle liste concatenate 320
PUSH, istruzione 462
PWB (Programmer's
Work Bench) 477

Q

QBX.EXE 84, 200
Quick BASIC
Opzione /L 91
QuickLibrary 476
QuickSort 347

R

radice
di un albero binario 327
registri di segmento 408, 433
Registro 10-12
16 bit 10
8 bit 10
bx 99
cx 99
dx 99
flag 13
RET, istruzione 471
RETRIEVE, istruzione 291
ricerca
in un albero binario 329
in una lista non ordinata 354
in una lista ordinata 360
ricorsione 347
ROM BASIC 395

S

salti condizionati
in linguaggio assembly 450
Scan Code 5
SEEK, istruzioni 295
Segmento 223
segmento di memoria 432
Set Breakpoint, comando 377-379
SETINDEX 278
shell sort 336
SHL e SHR, istruzioni 459
sistemi esperti 329
SS (Stack Segment) 433
stack 435, 462

strutture di dati 319
SUB, istruzione 448
SWAP, istruzione 343

T

tabella, numero di 285
Toggle Breakpoint, comando 367
TopC% 40, 46
TopR% 40, 46
tracepoint 384
TurnOffMenu( ) 158
TurnOnMenu( ) 158
TYPE, parola chiave 320

U

UIASM.LIB 85
UPDATE, istruzione 304

V

Variabile
Globale) 213
Locale 213
variabili 313

W

watchpoint 384
WORD PTR 495

THE PETER NORTON PROGRAMMING LIBRARY

BASIC AVANZATO

Steven Holzner e Peter Norton Computing, Inc.

Nel tempo, la programmazione non ha perso il suo fascino attirando verso di sé i consensi di numerosi nuovi utenti. Anche il BASIC, solitamente il primo linguaggio di programmazione, ha subito una naturale evoluzione che lo ha portato, in certi casi, ad uguagliare le prestazioni di linguaggi di rango superiore. Oggi con Microsoft QuickBASIC e Microsoft BASIC Professional Development System è possibile creare programmi stand alone. Il testo illustra le tecniche per aggiungere alle proprie applicazioni l'interfaccia grafica tipica e le prestazioni del software professionale.
Gli argomenti trattati e gli esempi forniti includono. gestione di finestre, menu pull-down, tecniche di database, routine di interfacciamento con linguaggio assembler, gestione dl mouse, grafica e animazione, interazione con DOS e BIOS.

Sommario

- Funzioni avanzate di I/O
- Interfaccia grafica a finestre
- Gestione del mouse
- Progettazione di menu pull-down
- Subroutine grafiche
- Tecniche avanzate di database
- Debugging
- interfacciamento con DOS, BIOS e linguaggio Assembler

Brady

GRUPPO EDITORIALE JACKSON

ISBN 88-256-0442-4

Cod. R1168